HEWLETT-PACKARD

Assembler ROM Manual

Owner's Manual

HP-87

HEWLETT PACKARD

Assembler ROM and HP 82928A System Monitor Reference Manual

HP-87

May 1982

ØØØ87-9Ø14Ø

Printed in U.S.A.

CONTENTS

INTRODUCTION

1.1 General Information

This manual outlines the commands, statements, instructions, and use of both the HP-87 Assembler ROM and the HP 82928A System Monitor. The manual assumes you have some knowledge of programming in assembly language. If you are not familiar with the HP-87 Personal Computer, you should read the owner's manual.

The HP-87 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 up to five binary (machine language) programs. The ROM contains the machine language program that recognizes and executes the statements provided by the BASIC language. Thus, the operating system ROM 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 external ROMs enrich the BASIC language by increasing the number of statements and functions that can be recognized and executed. Similarly, a binary program also extends the BASIC language.

1.2 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.
- Give increased execution speed.
- Redefine the system.

The Assembler ROM permits you to enter and edit source code for binary programs 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, 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 the mass storage device. The object code can also be loaded automatically or on command, and it is then ready to run.

To aid in programming, a disc is supplied with the Assembler ROM. This disc contains a global file of the system labels and their memory addresses for use during assembly. The disc also contains the sample programs from section 7 to help illustrate how binary programs are created and run.

The Assembler ROM gives you the ability to tailor statements for your own applications, to speed up program execution, and 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 computer. For this reason, you should understand assembly language programming before attempting to use the Assembler ROM.

1.3 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, 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.

System monitor instructions are discussed in detail in section 5 and the use of these instructions is demonstrated in section 7.

1.4 Using HP-83/85 Binary Programs on the HP-87

The HP-87 uses the same CPU as the HP-83/85. The programs are entered, stored, listed, and run in the same manner. There are some differences on the HP-87 which include:

- BASIC programs are stored in reverse order (executing from the higher addresses and progressing to the lower addresses).
- The extended memory controller makes it possible to access more memory.
- Five binary programs can be resident in the computer at a time.
- PTR1 is used as the BASIC program execution pointer at run time.
- PTR2 is used as the output stack pointer at parse time.
- Entire programs are no longer allocated before execution begins.
- The BASIC program control block in the HP-87 is 40 bytes long.
- The operating stack is of fixed length in the system RAM.
- String values are passed on the operating stack as a two-byte length and a three-byte address.
- Inverse video, more display modes, eight bit CRT addresses, and access during horizontal retrace periods are a few of the changes affecting the CRT.

Because the differences are only highlighted in this section, you should refer to individual sections in this manual to become familiar with the HP-87 Assembler ROM before writing programs.

To modify an existing HP-83/85 binary program for use on the HP-87:

1. Pick a binary program number and put it in the NAM statement. This should be a value between 200 and 377 (octal). Numbers from 0 to 177 are reserved for use by Hewlett-Packard.

   Two different binary programs may have the same binary program number, but they cannot be loaded and used at the same time. Attempting to do so will cause a BAD BIN-LOAD error.

2. Modify any ABS statements. In the HP-83/85, all binary programs were loaded so that the absolute base address could be calculated by the Assembler ROM at assembly time based upon the length of the binary program. In the HP-87, this is not true. A change must be made in the ABS pseudo-opcode. You must have an absolute base address.

   This only applies to binary programs that were written as absolute code. Most binary programs are relative and not affected by this change.

3. Modify all parse routines to use PTR2 as the output pointer rather than R12-R13.

4. Modify all parse routines to push the binary program token out as:

   ```
   TOK#   BPGM#   371
   ```

   rather than:

   ```
   371 GARBAGE-BYTE  TOK#
   ```

5. Change all RUNTIME references to R10 into references to PTR1.

6. Modify all code that uses string parameters that are passed on the R12 stack. These strings use three-byte addresses on the HP-87, rather than the two-byte addresses used by the HP-83/85.

7. Check all references to any system routines to see if any changes have occurred to the input/output conditions of the routine. Make any necessary changes.

8. Change all system/global address definitions.

9. Any routine that gets control through a RAM hook (such as CHIDLE, KYIDLE, IOTRFC) must calculate the base address of the binary program rather than loading it from BINTAB. Use the code:

   ```
   LABEL  LDM R20,R4
          BIN
          SBM R20,=LABEL
   ```

   This will leave R20-R21 with the absolute base address of the binary program. This change is necessary only in relative binary programs.

10. If the binary program uses its own error messages, ERRBP# (a RAM location in the system global addresses) must be set to the binary program number before calling ERROR or ERROR+.

1.5 Assembler Commands, Statements, and Functions

The commands and the statements and functions provided by the Assembler ROM are added those which are already part of the instruction set. They are executed exactly as the rest of the 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 6, and complete list may be found in sections 6 and 8.

Assembler Commands

A command is nonprogrammable, and can be executed only from the keyboard. 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.

ALOAD file name
Assembler Command

Legal only in assembler mode. Loads source code that was previously stored with the ASTORE command into computer memory from the file specified on the currently selected mass storage device. The file must be of the type known as extended **** or ASSM.

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 firmware and software may generate extended type files.

ASSEMBLE file name [,numeric value]
Assembler Command

Legal only in assembler mode. Assembles source code currently in the computer memory and stores it in the file specified by file name on the currently selected mass storage device. 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 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.

Note: 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, it should be stored there before typing ASSEMBLE.

ASSEMBLER
Assembler Command

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 function. The [CONT], [STEP], and [INIT] keys are inoperative in assembler mode. Displays READY when executed.

ASTORE file name
Assembler Command

Legal only in assembler mode. Stores the source code currently in the computer memory into the specified file on the currently selected mass storage device. File is of the type known as extended, shown in the directory as extended (****) or ASSM.

BASIC
Assembler Command

Legal only when in assembler mode, this command scratches memory and puts the computer back into BASIC mode. Display READY when executed.

FLABEL label
Assembler Command

Legal only in assembler mode. This command searches through the source code in memory for the label specified. For each occurrence of the label 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.

FREFS string
Assembler Command

Legal only in BASIC or assembler mode. Searches through the source code in memory for all occurrences of the specified string. After an FREFS command has been executed, pressing the [LIST] key causes the source code to be listed, beginning with the first line where the string occurred. Pressing any key will cause the FREFS command to halt prematurely.

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 computer. As with all BASIC statements and functions, they may be used either in calculator mode or as part of a BASIc program when in BASIC mode. When the computer is in assembler mode, all BASIC statements and functions may be executed only from the keyboard.

DEC
Assembler Provided BASIC Function

Returns the decimal equivalent of the specified octal value.

MEM address [:ROM#]],# of bytes][=#,#,..]
Assembler Provided BASIC Function

Dumps the contents of computer RAM or ROM memory to the current CRT IS device beginning with the octal address given. Continues dumping for the specified octal [,# of bytes]. At power-on, default # of bytes is 1ØØ octal; otherwise, default is the last # of bytes specified.

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

If =#,# is included in the statement, memory is not dumped, but instead the contents of memory locations beginning at the address given are changed to the octal values specified after the = sign. The memory locations must be in RAM. 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. Pressing any key will cause the memory dump to halt.

MEMD address [:ROM#][,# of bytes][=#,#,...]
Assembler Provided BASIC Statement

Same as MEM except reads the contents of three bytes of memory beginning with the address given and uses those contents as the address.

OCT decimal numeric value
Assembler Provided BASIC Statement

Returns the equivalent of the specified decimal value.

REL octal address
Assembler Provided BASIC Statement

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 last binary program that was accessed to yield the octal absolute addres. May be used alone or with the MEM command. May also be used with the command BKP if HP 82928A System Monitor is installed.

SCRATCHBIN
Assembler Provided BASIC Statement

Scratches all current binary programs from computer memory, without affecting anything else.

CPU STRUCTURE AND OPERATION

2.1 CPU Register Bank

The central processing unit (CPU) consists of 64 eight-bit registers, an address register pointer (ARP), a data register pointer (DRP), an arithmetic-logic unit (ALU), a shifter, and a set of status indicators.

The 64 eight-bit registers are grouped into two sections. The first 40 (octal) registers have two-byte boundaries and are used principally for addresses. Many of these bytes are reserved by the CPU for use as special purpose registers, and direct access to these should be avoided. The next 40 (octal) registers are separated by eight-byte boundaries. Floating-point numbers, 64 bits long, are stored here. The programmer must be aware of what is destroyed when the system uses these registers. The effects of system routines on register contents are found in section 8.

Any register in the CPU may be used as an accumulator when performing an operation. To distinguish between the registers, the CPU uses the DRP to designate the accumulator and the ARP to designate the operand. The DRP directs the results of arithmetic operations to the register it points to, and the ARP supplies the second operand when it is needed. 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.

Hardware-Dedicated Registers

Registers	Description
Ø,1	Register Bank Pointer: RØ points to the remainder of the CPU register bank. R1 is only accessable through RØ.
2,3	Index Scratch: R2-R3 are used for address calculation for indexed addressing.
4,5	Program Counter (PC): R4-R5 hold the absolute address of the next instruction location.
6,7	Return Stack Pointer: R6-R7 contain the pointer for the subroutine return stack. When a "JSB=" subroutine jump is executed, the CPU pushes the PC (R4-R5) on the stack. When the RTN is executed, the CPU pops two bytes from this stack and places them in R4-R5 (program counter).

Software-Dedicated Registers and EMC Pointers

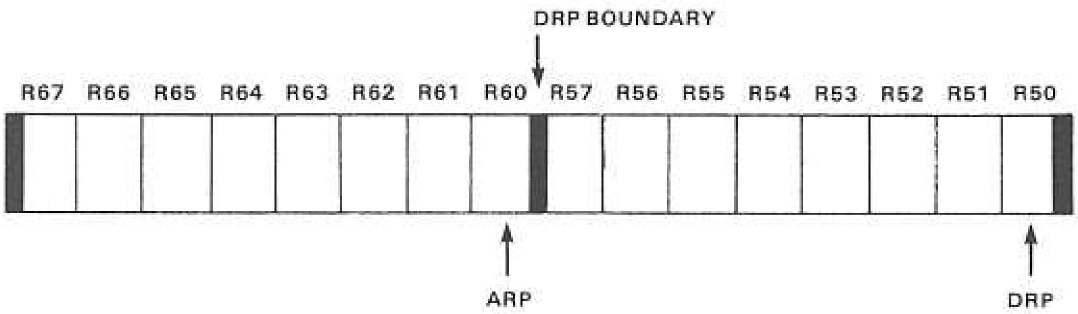
Registers/Pointers	Description
PTR1	At run time, contains the program counter (PCR), a pointer for executing BASIC programs.
PTR2	At parse time, used to point to the parse output stack.
1Ø,11	Not software dedicated at run time. When parsing, R1Ø-R11 point to the next character of the input ASCII stream.
12,13	Operation Stack: Parameters and results are passed on the stack pointed to by this register pair. Contains expressions when the BASIC program is decompiling.
14	When parsing or decompiling, R14 contains the current token being processed.

Software-Dedicated Registers and EMC Pointers

Registers/Pointers	Description
16	Current Status (CSTAT): R16 contains the code that indicates the current mode of operation. The table of CSTAT codes is found in paragraph 3.4.
17	External Communication Status (XCOM): When an external interrupt takes place the status is stored in R17. The table of XCOM status codes is found in section 3.

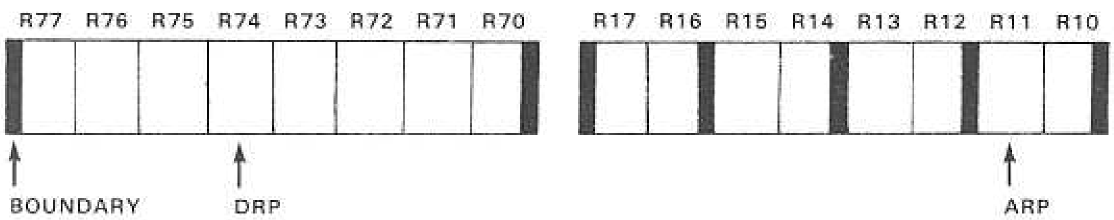
Multi-byte operations can be performed with the help of the register boundaries. The number of consecutive registers that will be used in the operation is determined by the distance between the DRP and the next boundary.

Example: In a multi-byte addition a 64-bit quantity contained in registers 5Ø through 57 will be added to a 64-bit quantity in registers 6Ø through 67.



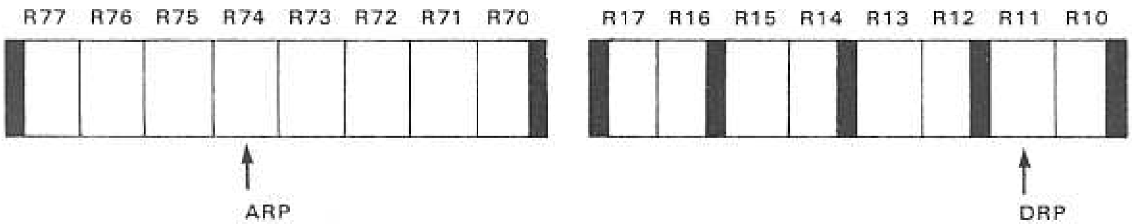
The operation begins with the registers pointed to by the DRP and the ARP, processing the registers within the boundary. The result is stored as a multi-byte quantity in the registers pointed to by the DRP.

Example: A multi-byte load with the DRP set to R74 and the ARP set to R11 will load the the four registers R74-R77 with the contents of R11-R14.



The boundary is determined by the DRP and is ignored by the ARP. In the previous example, the load terminates when the DRP reaches the next boundary.

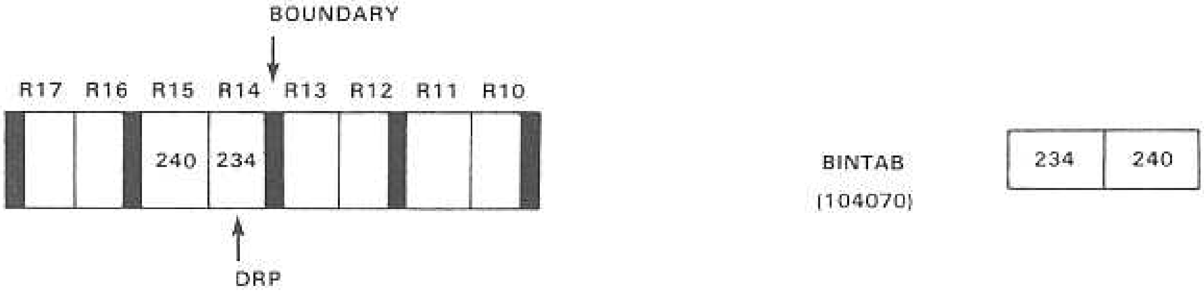
Example: The multi-byte store recognizes the boundaries in exactly the same way as the multi-byte load. Attempting to store with the DRP set to R11 and ARP set to R74 would result in the loss of several bytes due to the boundary after R11.



The boundary after R11 stops the multi-byte operation. Only one register is transferred to its destination, that is, R74. The DRP always determines how many bytes will be involved in a multi-byte operation.

There are also two-operand operations where the DRP points to one operand, and the second is located in computer memory. The number of bytes used in the operation is dependent upon the boundary after the DRP. That number of bytes of memory will be used starting at the location described by the label or pointer accessing computer memory.

Example: This load will be done with the address BINTAB, which is a label pointing to an address which contains the address of the start of the binary program. The DRP will point to R14.



Because the boundary is two bytes from the DRP, two bytes are accessed from memory.

2.2 Number Representation

The CPU can operate on numbers as octal and binary-coded decimal (BCD) quantities. All registers and register addresses are represented as octal numbers, and all floating-point numbers are represented in BCD notation, that is, each decimal digit is stored as a four-bit binary number, with two digits per register. Since the CPU cannot tell one representation from another, it is important to keep track of the way numbers are stored when doing arithmetic operations.

An address is always an octal value that occupies 16 bits or, for the extended memory pointers, 24 bits. The highest-numbered byte contains the first, or most significant, part of the address, and the lowest-numbered byte contains the last, or least significant, part of the address.

Example: The octal address 177605 is stored in two registers, R13 and R12. An address is always an octal value that occupies 16 bits and is contained in two registeres.

177605 = 1111111110000101

Binary Representation

R27 [1 1 1 1 1 1 1 1] R26 [1 0 0 0 0 1 0 1]

Octal Representation

R27 [3 7 7] R26 [2 0 5]

The ARP and the DRP will always point to the least significant byte of a multi-byte operation.

With BCD numbers, decimal one is represented with four bits, Ø Ø Ø 1, decimal two is Ø Ø 1 Ø, on up to decimal nine, which is 1 Ø Ø 1. When the decimal number has more than one digit, each digit is represented by four bits.

Example: The decimal number 3738 is represented by 16 bits.

3=ØØ11   7=Ø111   3=ØØ11   8=1ØØØ

Binary Representation

R27	Ø Ø 1 1 Ø 1 1 1	R26	Ø Ø 1 1 1 Ø Ø Ø

Octal Representation

R27	Ø 6 7	R26	Ø 7 Ø

Each byte can contain two four-bit BCD digits. Each register can represent numbers in the range ØØ to 99.

The ten's complement is used to simulate subtraction exactly like the two's complement is used in binary arithmetic. The ten's complement is formed by subtracting each binary-coded digit from nine (nine's complement arithmetic), 1 Ø Ø 1, then putting the digits back together to form the number again and finally incrementing the entire quantity by one (one's complement arithmetic).

The negative of a number in BCD representation, for subtraction purposes or in special cases to show the sign of an exponent, is found by taking the ten's complement.

Example: To find the negative of 19, each digit, 1 and 9, is subtracted from 9, or, another way of looking at it, 19 is subtracted from 99.

```
   99        1ØØ1 1ØØ1
 - 19      - ØØØ1 1ØØ1
   8Ø        1ØØØ ØØØØ
```

Add one to the combined result:

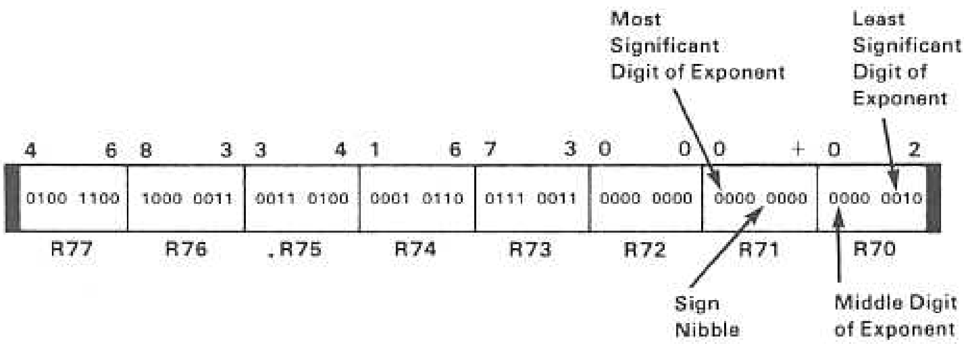
```
  80          1000 0000
+ 01        + 0000 0001
  81          1000 0001
```

In effect, when 81 is added to 19 the result is 00 in BCD notation.

Numeric quantities may be represented as real floating-point, short, and integer formats. The real and short forms are expressed as BCD digits, and the integer form is a five-digit number with a sign digit at the end of the quantity. The system represents all numeric quantities in BCD notation.

Real numbers have a mantissa of 12 digits, and exponent and sign information, all stored in eight bytes. The mantissa fills the 12 most significant nibbles of the number, the sign takes one nibble, and the exponent is contained in the last three nibbles. The most significant digit of the number is stored in the most significant byte, and the decimal point is assumed to be immediately after the most significant digit. The sign of the number follows the least significant digit of the mantissa, and the exponent, expressed in ten's complement notation, is found in the three least significant nibbles of the quantity.

Example: The real number 468.3341673 (in scientific notation 4.683341673 x 10^2), would be represented in BCD as:



The radix is assumed to be in R77 between the four and the six.

Integers are stored in three bytes, with five digits and a sign. The most significant digit of the integer is stored in the least significant byte. Real and short number representations are not right justified like integer representation.

Example: The integer 6483 would be represented in BCD notation from the least significant digit, 3, to the most significant digit, 6, with the sign, positive or Ø Ø Ø Ø, in the most significant four bits of the quantity.

+ 0	6 4	8 3					
0000 0000	0110 0100	1000 0011					
R77	R76	R75	R74	R73	R72	R71	R70

Short numbers have a mantissa of only five digits and an exponent of two digits. Both the mantissa and the exponent have sign bits, found in the most significant digit. The representation of the mantissa begins immediately following the sign bits with the most significant digit of the mantissa found in the second digit of the most significant byte. The assumed decimal point is directly after the first digit, then the rest of the mantissa is represented. The two least significant digits hold the exponent, which is not in complement form because the sign of the exponent is in the most significant digit.

Example: The short number -.ØØ64 need not be represented as a 12-digit real number. In BCD short form it is represented as:

– 6	4 0	0 0	0 3				
0011 0110	0100 0000	0000 0000	0000 0011				
R77	R76	R75	R74	R73	R72	R71	R70

The radix is assumed to be between R77 and R76.

2.3 Status Indicators

The 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. The instruction set has data movement instructions of both the arithmetic and nonarithmetic types. These instructions include:

- Arithmetic: Add, substract, compare, increment, decrement, and complement.
- Nonarithmetic: Load, store, "and", "or", "exclusive or", shift, clear, and test.

The CPU contains the following one-bit status flags and four-bit extend register:

DCM Decimal Mode Flag: This flag determines whether the system is using binary numbers or BCD numbers in arithmetic operations. In BCD mode, each decimal digit is converted to BCD, and all arithmetic operations are done with the resulting four-bit digits. This is the way floating-point real numbers and integers from BASIC programs are represented.

The system uses this flag to determine the correct mode, so the user must make sure it is set properly for arithmetic operations. All shifts and all arithmetic operations are affected by the DCM flag.

Two instructions affect the status of the DCM flag: BCD sets it to 1, and BIN clears it.

E Extend Register: In BCD mode,this four-bit register will accept the displaced digit resulting from a shift. Once in the register, a BCD digit may be incremented, decremented, or cleared, and, if needed, the digit may be returned to the register it came from using the extended shift instructions.

CY Carry Flag: In binary mode, this flag will indicate the result of a bit shift. A bit may be shifted into the CY flag, tested, and then shifted back into a register, using the extended shifts. It functions similar to the extend register in BCD mode.

During all arithmetic operations, the CY flag will be set with the carry out of the most significant part of the operation. In addition between two numbers where the result is too large for the register to hold, or in subtraction where the result is positive, the CY flag is set to 1. The CY flag may be thought of as the "borrow" if needed for subtraction.

When two quantities are added, the CY flag is set with the carry, if any, resulting from the addition of the most significant bits.

Examples:

If two positive numbers, both with a most significant bit of 0, are added, then the carry will always be 0.

```
             0 1 0 0 0 0 1 0
           + 0 0 1 0 0 1 0 1
   CY  0     0 1 1 0 0 1 1 1
```

If a positive number is added to a negative number, in reality a subtraction, then two possibilities could occur:

1. The result could be negative, in which case no carry would be made.

```
             0 0 1 0 1 1 1 1
           + 1 0 0 1 0 1 1 1
   CY  0     1 1 0 0 0 1 1 0
```

2. The result could be positive, causing a carry out.

```
             0 1 0 0 1 1 0 1
           + 1 1 1 0 1 0 0 0
   CY  1     0 0 1 1 0 1 0 1
```

If two negative numbers are added then the CY flag is set to 1.

```
         1 1 0 0 0 0 0 0
       + 1 0 0 0 0 0 1 0
CY [1]   0 1 0 0 0 0 1 0
```

The carry flag is set by comparisons in the same manner as additions.

An increment sets the CY flag if the data register is all 1's.

```
         1 1 1 1 1 1 1 1
       + 0 0 0 0 0 0 0 1
CY [1]   0 0 0 0 0 0 0 0
```

OV Overflow: The overflow status is determined by taking the "exclusive or" of the CY flag and the most significant bit of the data register. It is set to 1 when the addition of two positive numbers yields a negative result, when the addition of two negative numbers yields a positive result, and when the result of a left shift changes the sign of the data register.

```
         0 0 1 0 1 1 1 1        057
       + 0 1 1 0 0 1 0 1      + 145
OV [1]   1 0 0 1 0 1 0 0        202
```

OD Least Significant Bit: After any data movement instruction, the least significant bit is shown as the OD flag. If the OD flag is set to 1, then the number is odd. If the OD flag is 0, the number is even. The right-most bit in the data register is always the least significant bit.

```
0 0 1 0 0 1 1 0   [0] OD
```

NG Most Significant Bit: This flag displays the most significant bit in the data register. If this flag is set to 1 then the quantity is negative, and if the NG flag is clear then the quantity is positive.

NG [Ø] [Ø Ø 1 Ø Ø 1 1 Ø]

ZR Zero: If the data register is Ø or if a comparison is made between two equal numbers then this flag is set to 1.

[Ø Ø Ø Ø Ø Ø Ø Ø] [1] ZR

LDZ Left Digit Zero: This flag is set if the left-most four bits are Ø Ø Ø Ø. In BCD mode this would indicate the most significant digit.

LDZ [1] [Ø Ø Ø Ø 1 Ø 1 1]

RDZ Right Digit Zero: If the least significant four bits are Ø Ø Ø Ø then this flag is set to 1. In BCD mode this would indicate the least significant digit.

Example: Status information is based on the entire multi-byte quantity that is being processed. All multi-byte operations, except right shift, start execution with the least significant byte. The right shift starts with the most significant byte. All status flags, except OD, RDZ, and DCM, are updated after each byte of execution and will be correct as the register boundary is met. The OD and RDZ flags are set for the first byte and never changed. The E, CY, and OVF flags are only affected by arithmetic operations.

After the multi-byte addition of the two system addresses, OFFSET (ØØØ1ØØ) and the label VARIABLE (ØØØ365), the status indicators will be set as follows:

OFFSET	1 Ø Ø Ø 1 Ø Ø Ø	Ø Ø 1 1 1 Ø Ø Ø
+ VARIABLE	Ø Ø Ø Ø Ø Ø Ø Ø	1 1 1 1 Ø 1 Ø 1
RESULT	1 Ø Ø Ø 1 Ø Ø 1	Ø Ø 1 Ø 1 1 Ø 1

DCM	E	CY	OV	NG	OD	LDZ	RDZ	ZR
Ø	ØØØØ	Ø	Ø	1	1	Ø	Ø	Ø

OPERATING SYSTEM

3.1 Introduction

This section explains how system memory is allocated, how extended memory is accessed, 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.

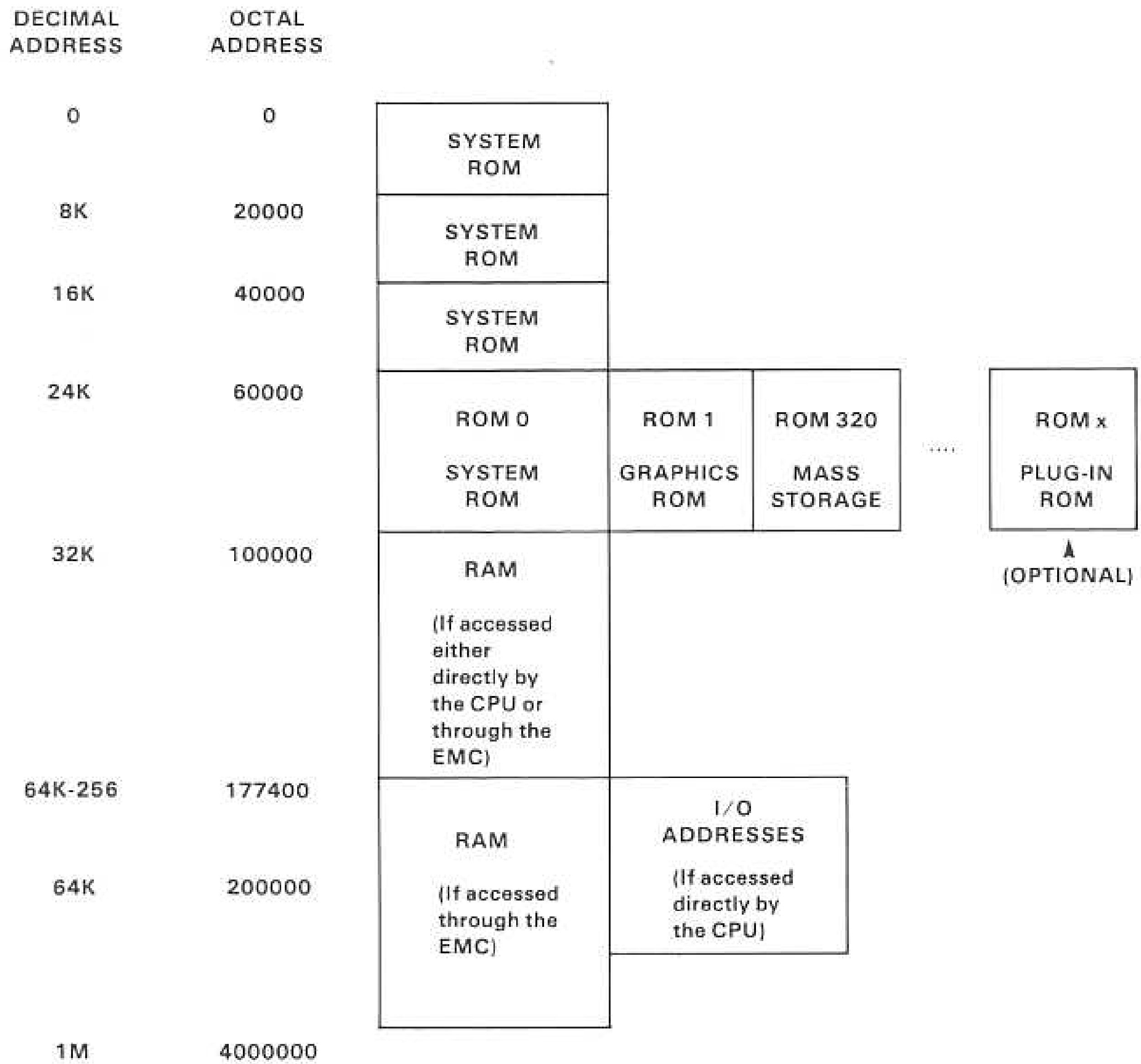
BASIC programs are executed by an interpreter. However, the code that is interpreted is vastly different from the BASIC statements as they were originally entered. As the statements are entered, they are parsed and compiled into a form of RPN (Reverse Polish Notation), which can be interpreted more efficiently. The BASIC reserved words are converted to single-byte tokens (refer to Execution by Tokens). This makes the internal form of the code somewhat more compact than the original form, and also makes interpretation easier and faster.

Also during the process of parsing and compiling, variables are placed in a variable storage area, with only their addresses and names remaining in the area containing the tokens. The BASIC program is held in memory as a series of tokens and addresses of variables and associated data bytes. To execute the program, the computer processes these token and variable addresses in order. As each token is processed, it causes the machine to access a table of routine addresses and execute a specific routine corresponding to the token. If the token indicates a variable, the machine uses the next three bytes as the variable address.

3.2 System Memory

Several distinctly different regions comprise the system memory. They are (all numbers are octal unless indicated otherwise):

- Six system ROMs, each containing 8192 decimal bytes. A subset of the ROM area is the address range from 60000 to 77777. This range is shared by system ROMs 0, 1, 320, and all of the external plug-in ROMs. Each of the ROMs in this area can be selected or deselected for talking on the bus, but only one of them can be selected at a time. Each of these ROMs has a bank-select address which is its ROM number, ranging from 0 to 376. To select a particular ROM you store the desired ROM number to an I/O address called RSELEC. The chosen ROM will be selected and all other bank-selectable ROMs will be deselected.
- RAM, 32768 (decimal) bytes in the basic machine.
- Memory addressable directly by the CPU (addresses 0 to 177777).
- Memory addressable through the extended memory controller (EMC), 32K-544K.
- The block of 400 addresses (177400 to 177777) which act as I/O addresses, when accessed directly by the CPU. The same addresses accessed through the extended memory controller will act as RAM memory, not as I/O addresses.

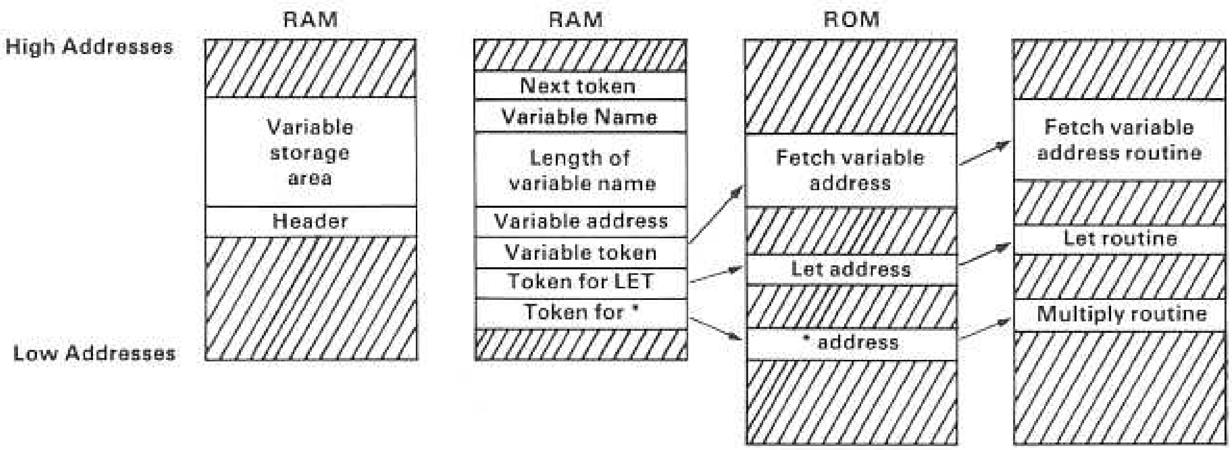

DECIMAL ADDRESS
OCTAL ADDRESS
0
0
SYSTEM ROM
8K
20000
SYSTEM ROM
16K
40000
SYSTEM ROM
24K
60000
ROM 0
SYSTEM ROM
ROM 1
GRAPHICS ROM
ROM 320
MASS STORAGE
ROM x
PLUG-IN ROM
(OPTIONAL)
32K
100000
RAM
(If accessed either directly by the CPU or through the EMC)
64K-256
177400
RAM
(If accessed through the EMC)
I/O ADDRESSES
(If accessed directly by the CPU)
64K
200000
1M
4000000

Computer Operation

The basic machine 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 8.

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 memory. At certain times in the operation of the system, the resident binary programs and ROMs are polled by the main system. In addition, there are a number of entry points (hooks) that allow operation to be intercepted and modified by a binary program or ROM. These hooks are normally idle, but they can be used to take over the system at certain key times.

Execution by Tokens (Run Time)



Tokens are used to represent the keyword, such as LET, FOR, BEEP, etc., that make up each BASIC statement. Each token is a one-byte quantity that the machines uses to find the addresses of routines associated with that token. Each token must have an associated entry in a table of routines for execution at run time, another entry in an ASCII keyword table, and a third entry in a table of parse routines. A list of system tokens may be found in section 8.

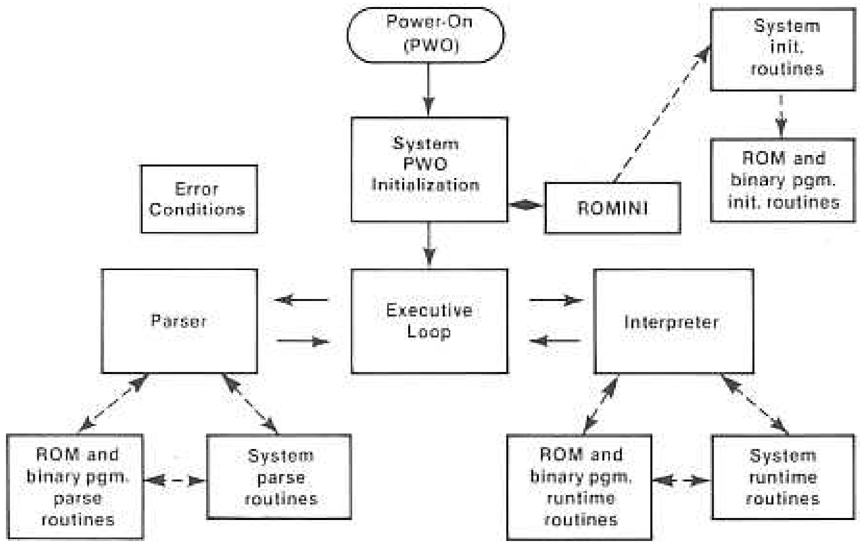
The computer is a token-driven machine. A program is held in memory as a series of tokens and variable addresses which the machine processes.

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

Some tokens indicate to the system that the three bytes following the token contain a variable address. In this case, the system attempts to find the variable in the variable storage area and, if not found, creates a place for it. Other tokens indicate that the bytes following the token are constants to be pushed onto the R12 stack.

Two tokens, 370 (octal) and 371 (octal), 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 number is the token within the ROMs table that is to be executed. Token 371 directs the system to a binary program in the same way.

3.3 Overall System Flow



System flow is shown by the chart above. In general, loading and running a program, or executing a calculator mode statement, will require execution within the following areas:

Power-on Initialization: When the computer initially powers-up, it performs a sequence of operations: performs a self-test, accesses and resets any interface modules, reserves memory for later use, allows any ROMs to reserve memory, and returns to the system.

Executive Loop: External stimulus (such as a keyboard interrupt) and changes within the computer (such as an error condition) will cause the executive loop to call the appropriate routines to take control at the right time.

Parser: Parsing occurs when [END LINE] is pressed after a program line or calculator mode statement has been typed. Parsing is the changing of ASCII code into tokens. The parser first searches the ASCII tables in any resident binary programs for a keyword match, then the ASCII tables in any external ROMs, and finally the system tables. This makes it possible to redefine system keywords.

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

In addition, 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 routines in the resident binary programs last. ROMFL is a RAM location that initialization routines called by ROMINI can look at to see why they were called.

Initialization routines are called before, during, or after a condition occurs, depending upon the following conditions:

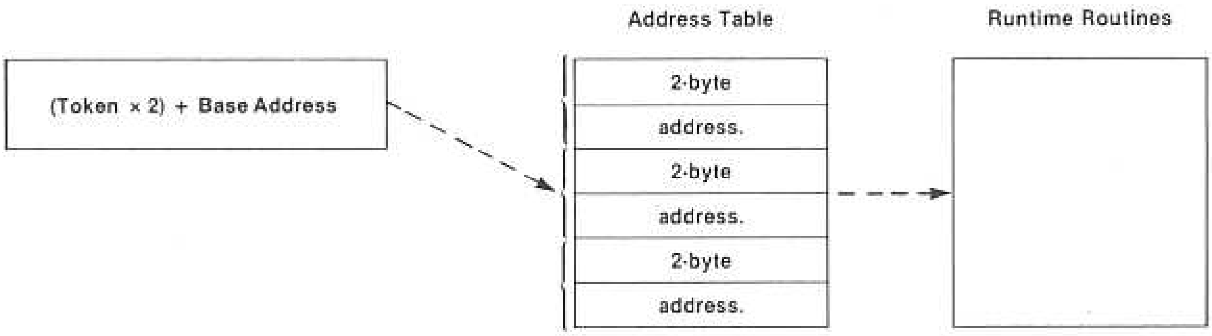
ROMFL	Meaning	Initialization Routines Called
Ø	Power-On	After system initialization.
1	RESET	After system reset.
2	SCRATCH	Before scratch.
3	LOADBIN	After loadbin.
4	RUN	Before execution begins.
	INIT	After allocation done.
5	LOAD	Before load.
6	STOP, PAUSE	During.
7	CHAIN	After.
1Ø	Allocate token class>56	During.
11	Deallocate token class>56	During.
12	Decompile token classs>56	During.
13	Program halt on error	During.

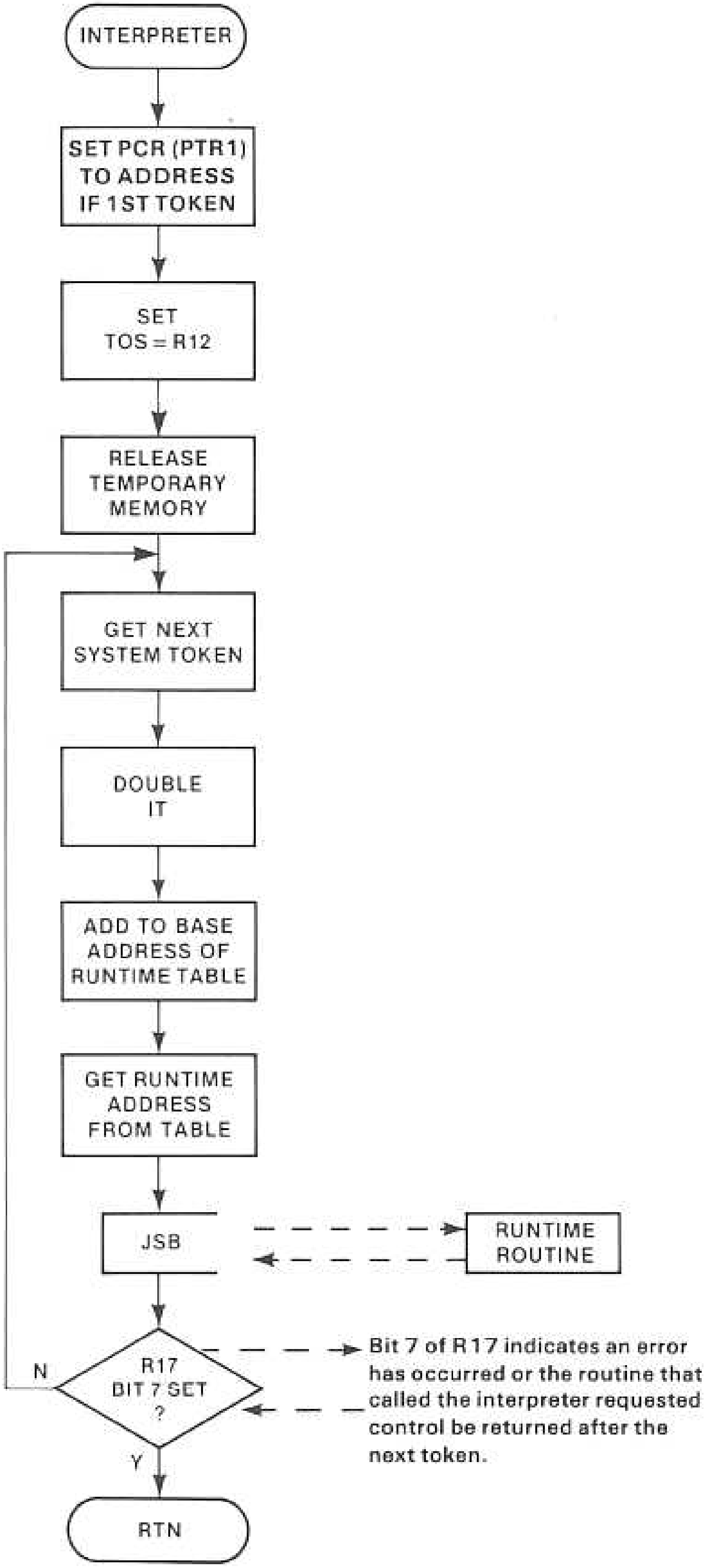
When errors occur, the system generates the proper warning or error message.

Interpreter Loop

The interpreter loop fetches the next token, processes it, and passes control to the respective run time code. When the run time code has been executed, control returns and the interpreter continues with another token.

A token is an ordinal into a table of addresses. The address table is made up of two-byte addresses. 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 current address.





INTERPRETER LOOP

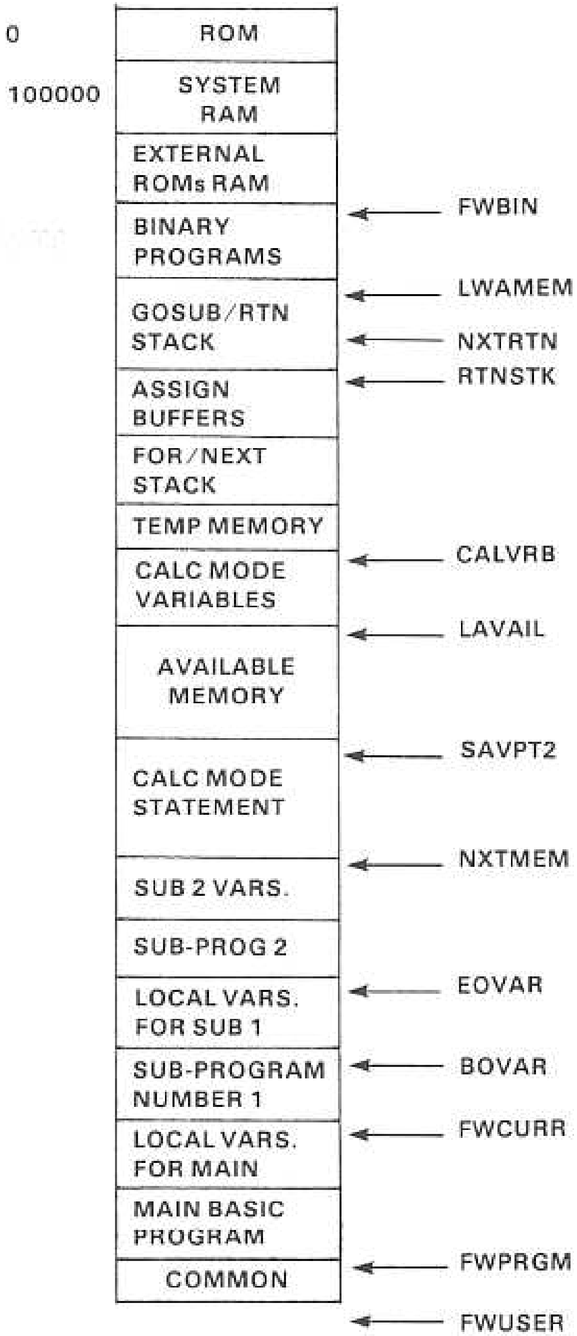
3.4 Allocation and Deallocation

Allocation is the process of reserving and assigning memory for program variables. The three modes of allocation are:

- If the program line is a dimension statement, the entire line is allocated before execution continues.
- If the current token being allocated is the start of a user defined function (DEF FN), then allocation will continue for the duration of the definition (until the FN END) before execution will resume.
- All other tokens are allocated one at a time as they're encountered.

The class of a token determines if the token needs to be allocated. The following diagram shows how memory looks in a program that has been allocated:

RUNTIME



The GOSUB/RTN stack is for the BASIC program not for assembly language. NXTRTN points to the next RETURN address on the stack.

Each ASSIGN buffer takes 284 bytes.

TEMP memory is released by the system at the end of each line of a program and when an "@" token (statement concatenation) is encountered.

Both the RUN and the CONT commands set LAVAIL equal to CALVRB, so during the running of a BASIC program, they will always be equal, and there will be no CALC mode variable.

If a CALC mode statement has been entered and is executing, it will begin at NXTMEM − 1 and end at SAVPT2. Otherwise, SAVPT2 will be equal to NXTMEM.

NXTMEM points to the last byte of the BASIC programs (or sub-programs).

EOVAR points to the last byte of the variable space of the current active BASIC program and BOVAR points to the [first byte] + 1.

FWCURR points to the first byte of the currently active program.

FWPRGM points to the first byte of the MAIN BASIC program.

FWUSER points to one higher than the highest address in memory.

Unless you have created a token whose class is greater than 56, allocation is handled by the computer. Because a program is allocated in segments, a memory overflow could occur after you are well into your program.

The system executes the BASIC program starting at the highest address. The line of BASIC code

```
10 A=B
```

would be parsed into this stream of bytes:

```
016       END OF LINE TOKEN
010       STORE SIMPLE VARIABLE TOKEN
102       B, THE VARIABLE NAME
001       LEN OF VARIABLE NAME
000 )
000 }     3 BYTES RESERVED FOR ALLOCATION TIME ADDRESS
000 )
001       FETCH SIMPLE NUMERIC VARIABLE TOKEN
101       A, THE VARIABLE NAME
001       LEN OF VARIABLE NAME
000 )
000 }     3 BYTES RESERVED FOR ALLOCATION TIME ADDRESS
000 )
021       STORE SIMPLE NUMERIC VARIABLE TOKEN
016       LEN OF LINE (16 OCTAL BYTES FOLLOW)
020 )
000 }     BCD LINE NUMBER 10
000 )
```

Token number 21 (and tokens such as 1, 2, 3, 22, and 23) is immediately followed by three bytes which are used to contain the relative address from the first byte of the currently active program (FWCURR). Since the variable storage area for BASIC programs is at a lower address than the program, this relative address will always be negative (that is, the most significant bit of the address will always be set). Therefore, if the most significant bit is 0 then the system knows that the current token has not yet been allocated. In this case, the allocator would be called, which would search through the variable storage area for the current variable. If found, the allocator would calculate the correct relative address and place it where the three 0's are following the 21 token. If not found, the allocator would create a storage area for it in the space allocated for variable storage (EOVAR), then calculate and store the relative address after the number 21 token. Execution would then continue.

Since the variable name could be long or short, the length of the name and the ASCII characters for the name immediately follow the address. The length of the name and the ASCII characters will be skipped at execution time, similar to the way a comment is skipped.

Line numbers are handled in a similar manner. The BASIC code

```
10 GOSUB 100
```

would be parsed like this:

```
016     END OF LINE TOKEN
000 )
001 }   BCD LINE NUMBER OF DESTINATION
000 )
133     GOSUB LINE NUMBER TOKEN
005     LEN OF LINE
020 )
000 }   BCD LINE NUMBER 10
000 )
```

Since line numbers are only five digits long, the most significant bit will be 0 if the line is not allocated. If the line is allocated the address will always be negative and the most significant bit will be set. All line numbers are converted into addresses relative to FWCURR (the first byte of the currently active program) at allocation time.

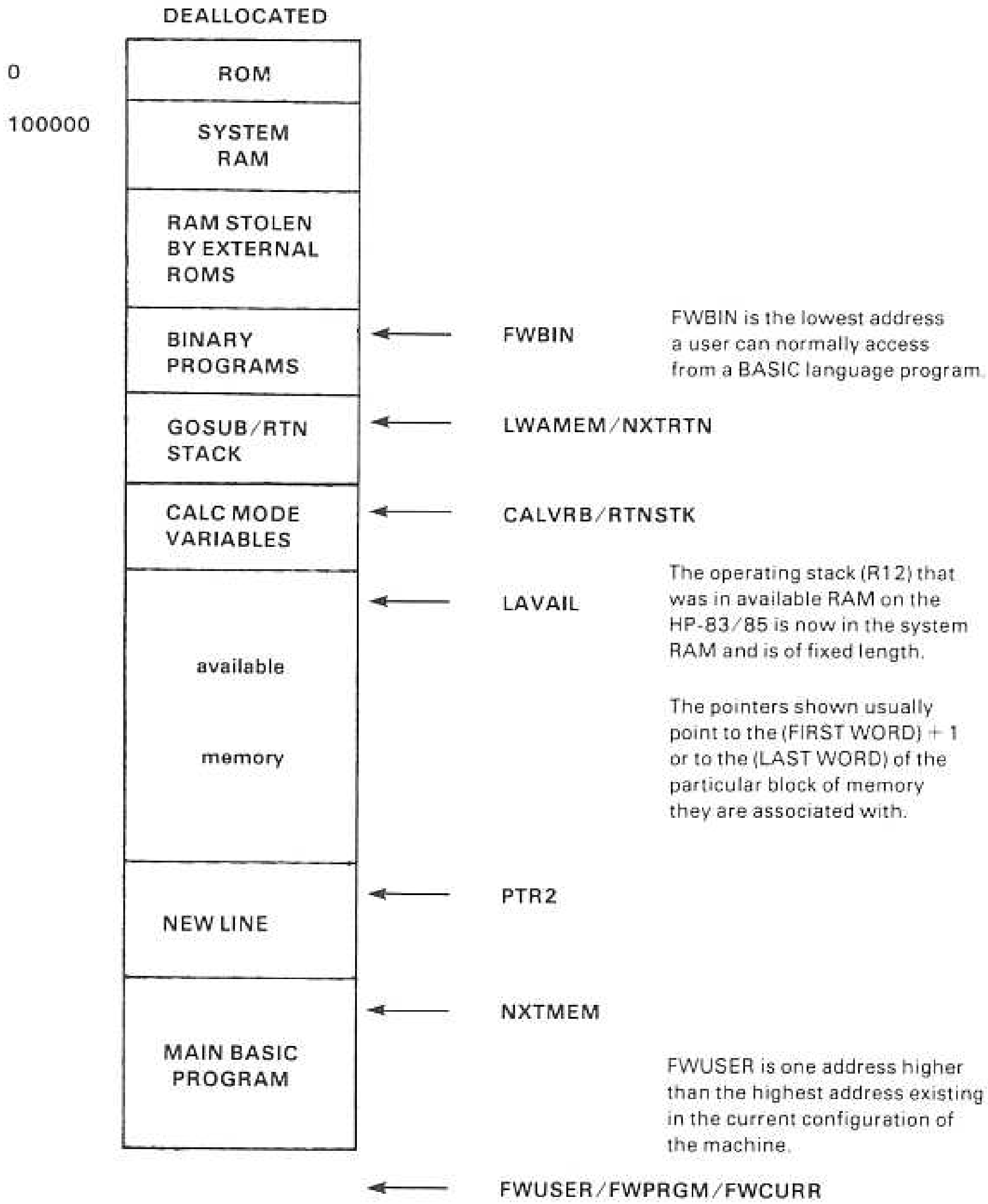
Line labels are handled in a way similar to variable names. The BASIC code

```
10 GOSUB [linelabel]
```

is parsed as:

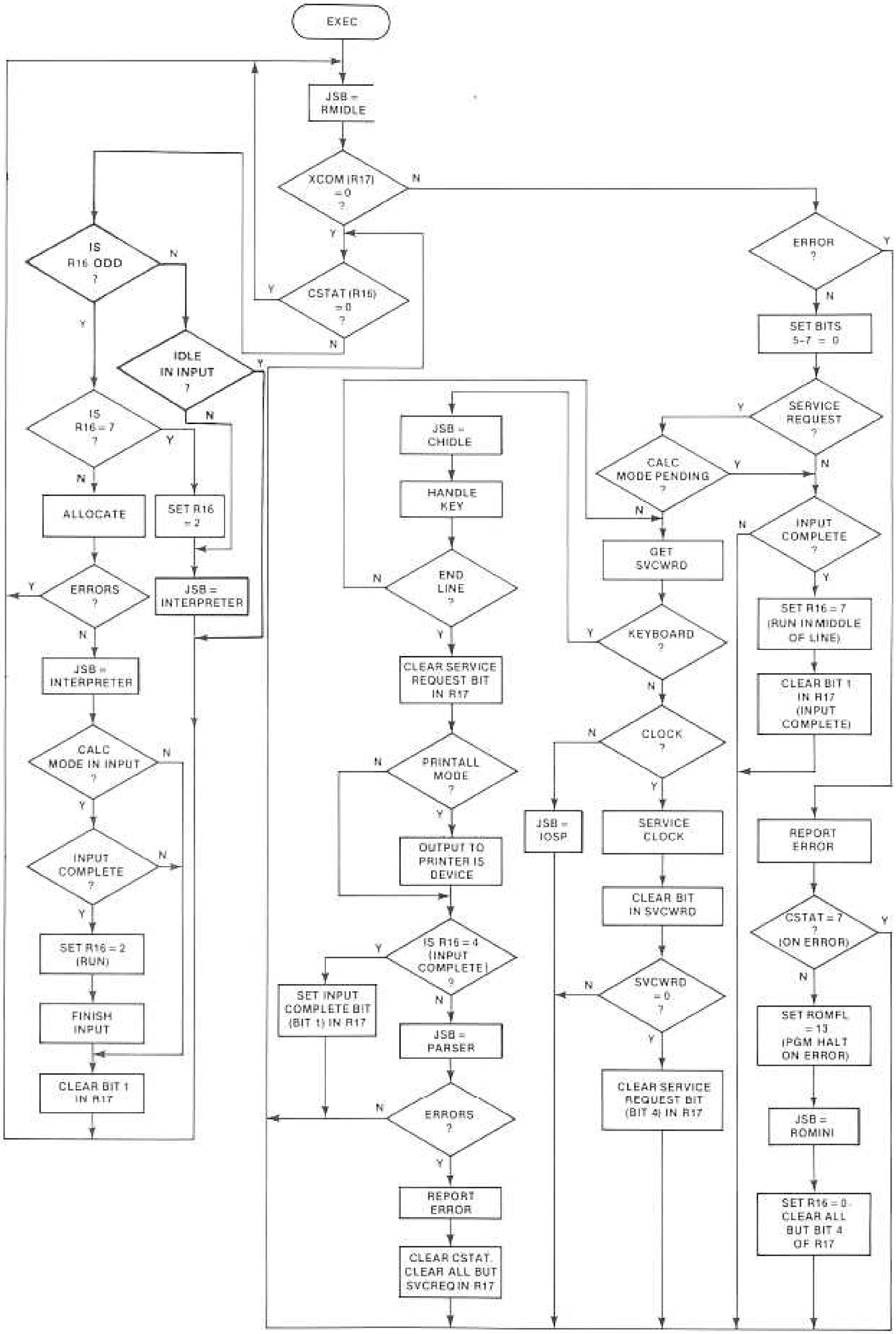
016	END OF LINE TOKEN
154	l
145	e
142	b
141	a
154	l
145	e
156	n
151	i
114	L
011	LEN OF LINE LABEL
000	3 BYTES RESERVED FOR ALLOCATION ADDRESS
000	
000	
270	GOSUB line label TOKEN
017	LEN OF LINE
020	BCD LINE NUMBER 10
000	
000	

A program is deallocated while you are typing in lines or while it is being edited. When a BASIC statement is typed and [END LINE] is pressed, the computer deallocates the program if it has not already been done. Program variables are held as names rather than addresses. This diagram shows memory when a program is deallocated:



3.5 Executive Loop

After power-on initialization, the executive loop portion of the system takes control. The executive loop examines CPU registers R16 and R17 for changes in the status of the computer, listens for external communications, and takes the appropriate actions based upon the information received. The current status information (CSTAT) is kept in register R16 and the external communication flags (XCOM) are kept in register R17. As long as registers R16 and R17 are both zero, the system is idle. The executive loop flowchart is shown on the next page.



EXECUTIVE LOOP

3.6 Interrupts

When there is a change in status and the system is no longer idle, CSTAT (R16) indicates the computer mode of operation, according to the value stored there.

CSTAT

Value	Current Status
Ø	Idle.
1	Calculator mode execution.
2	Program is running.
3	Not used.
4	Idle during input statement.
5	Calculating during input statement.
6	Not used.
7	RUN in middle of a line.
8-255	Not used.

If execution halts, the system needs to know what caused it to halt and how to handle it. Each of the eight bits in XCOM (R17) have a different meaning associated with it. The service request bit is the only bit directly affecting interrupts.

XCOM

Bit Set								Execution Halt
7	6	5	4	3	2	1	Ø	
							X	End of calculator mode.
						X		Input complete.
					X			Step mode.
				X				Trace mode.
			X					Service request (any interrupt).
		X						Immediate set.
	X							Error set.
X								Break ("or" of bits 5 and 6).

One of the controlling devices on the internal communications bus will generate an interrupt to begin execution. An interrupt will set bit 4 in R17 (XCOM) and a bit in a memory location which is used to keep track of the cause of an interrupt (SVCWRD). The executive loop knows that the interrupt has occurred (from XCOM) and which device caused the interrupt (from SVCWRD).

SVCWRD Bit Set 7	6	5	4	3	2	1	Ø	Type of Interrupt
							X	Keyboard interrupt.
						X		I/O interrupt.
					X			Timer 1 interrupt.
				X				Timer 2 interrupt.
			X					Timer 3 interrupt.
		X						Special interrupt.
	X							Not used.
X								Not used.

Whenever an interrupt occurs, the CPU expects the interrupting device to send a pointer to an interrupt handling routine in a table of addresses. This pointer is a one-byte quantity and the two bytes that it points to in memory indicate the starting address of the service routine. If multiple interrupts occur then the first interrupt is handled and the rest are disabled.

The service routine pointers are located at addresses Ø thru 25 in memory.

Table of System Interrupt Pointers

ADDRESS	CODE	FUNCTION
ØØØØØØ	DEF STARTX	Power-on vector.
ØØØØØ2	DEF SPARØ	Spare hook Ø.
ØØØØØ4	DEF KEYSRV	Keyboard.
ØØØØØ6	DEF SPARØ	Spare hook Ø.
ØØØØ1Ø	DEF CLKSRØ	Clock Ø.
ØØØØ12	DEF CLKSR1	Clock 1.
ØØØØ14	DEF CLKSR2	Clock 2.
ØØØØ16	DEF CLKSR3	Clock 3.
ØØØØ2Ø	DEF IRQ2Ø	I/O modules.
ØØØØ22	DEF SPAR1	Spare hook 1.
ØØØØ24	DEF SPAR1	Spare hook 1.

An interrupt may be caused by the keyboard, a timer, an I/O module, or a special device. Keyboard interrupts are handled using KEYSRV and the character editor (CHEDIT). If the clock causes an interrupt, an ON TIMER routine is called. An interrupt from an I/O module is handled by the IRQ2Ø and IOSP hooks, and special interrupts must be handled by the spare interrupt routines SPARØ and SPAR1 from other hardware.

Programmer created interrupt routines may be handled by taking control of certain memory locations accessed by the executive loop or by taking control of the interrupt service hooks SPARØ, SPAR1, KYIDLE, or IRQ2Ø. The interrupt service hooks are accessed prior to the executive loop. Therefore, these locations may bypass the executive loop. Jumps to these locations (hooks), cause the instructions located there to be executed. Initially, only a RTN instruction is stored at each of these locations, so control immediately passes back to the executive loop.

When an I/O device interrupts the system, a jump is made to IRQ2Ø before control passes to the executive loop. This gives the I/O interrupt routine the chance to bypass the operation of the executive loop, taking control more efficiently.

The executive loop always performs these functions: tests CSTAT, tests XCOM, and jumps to RMIDLE. If an interrupt has occurred from the keyboard, a jump is made to CHIDLE. When an I/O interrupt occurs a jump is made to IOSP, provided that the proper bits in XCOM and SVCWRD are set.

When an interrupt occurs during the execution of a program, the CPU finishes the current instruction, saves the program counter (R4-R5) on the R6-R7 stack, and acknowledges the interrupt. The device puts a pointer to the address of the service routine on the bus, and the CPU loads the service routine address into the program counter (R4-R5). This is effectively a subroutine jump to the service routine, because the return address has been saved. The status of the CPU and the contents of any registers that will be used in the service routine must be saved and restored from within the routine. This is important because an interrupt could occur between the execution of an instruction which sets the status indicators and an instruction that depends on that status.

3.7 Hooks

A binary program or a ROM can gain control of the system using RAM hooks. Some are accessed directly by the executive loop and some by routines that branch from the executive loop. The four types of hooks are:

- Language hooks: Allow you to create new BASIC keywords or redefine existing ones.
- General hooks: Allow you to take over various parts of the operating system by storing subroutine jumps to a binary program or ROM routine at specified RAM locations.
- Initialization routines: Called by the system, external ROMs, and binary programs at initialization time. An initialization routine can steal RAM, change flag status, or gain control of the operating system.
- Error message table: Allows a binary program or ROM to flag specialized error conditions with custom error messages.

Language Hooks

With language hooks the binary program or ROM can define new keywords, functions, or auxiliary tokens. Because the system first polls the resident binary program and then all external ROMs, a binary or ROM program can take over or supersede the system tables.

General Hooks

To provide for each general hook, the system at certain times executes a subroutine jump to a specific RAM location. During normal operation each of these RAM locations contains a RTN or is otherwise idle. By placing a jump 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 external program writer 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.

Because support is not supplied by the system before calling any of the RAM hooks, any binary program base address might be in BINTAB when the system calls a hook. You must ensure that the correct base address is loaded into BINTAB before a hook is taken.

The following code stores a copy of the binary base address for future use:

```
**Initialization Routine**
INIT   LDBD R34,=ROMFL        See why INIT routine was called.
       BIN                    Binary mode necessary for CMB and
                              ADM instructions.
       CMB R34,=3             Is a binary program being loaded?
       JNZ INITRTN            If no, return.
       LDMD R34,=BINTAB       If yes, save the binary base address.
       STMD R34,X34,OURBAS    Store it in the program.
       ADM R34,=HOOK          Make hook routine address absolute.
       STM R34,R45            Make a copy of the address to store
                              in HOOK.
       LDB R47,=236           Load the opcode for return instruction.
       LDB R44,=316           Load the opcode for a JSB instruction.
       STMD R44,=CHIDLE       Store R44-R47 into CHIDLE.
**Hook Routine**
INIRTN RTN                    Done.
HOOK   BIN                    Entry to hook routine.
       DRP R34                Set the DRP to R34.
       BYT 251                Do a LDM R34,= instruction.
**Store Base Address Here**
OURBAS BSZ 2                  Base address is stored here.
       STMD R34,=BINTAB       Load the base address into BINTAB.
```

Unless otherwise noted, each general hook is seven bytes long. Flowcharts are provided for selected hooks in section 8 of this manual. General hooks are supplied at the following points:

RAM Name	Location	Function
CHIDLE	103670	Character editor intercept.
DCIDLE	104035	System decompiler hook called at entry time. If you take this hook and don't want to let the system have a chance at decompiling, then you need to discard a couple of return addresses.
DGHOOK	104044	If the PLOTTER IS select code is one or two and a DIGITIZE command is executed, this hook will be called so software that has been loaded can digitize off of the CRT.
IMERR	103724	Used to expand the IMAGE statement. This hook is called when there is something in an IMAGE statement that the system doesn't recognize.

RAM Name	Location	Function
IOSP	103652	I/O service pointer. Used by I/O and mass storage ROMs.
IOTRFC	103643	General output hook. If the select code of the CRT or PRINTER IS device is not 1 or 2, the DISP or PRINT will go to IOTRFC.
IRQ20	103742	The CPU vectors to IRQ20 when an I/O module interrupts.
KYIDLE	102425	Keyboard intercept. Polled whenever a key is pressed.
MSHIGH	103764	High level hook that allows modification of mass storage commands.
MSLOW	103773	Low level hook to allow driving of mass storage devices not already supported by the system mass storage ROM.
MSTIME	104002	TIMEOUT hook in the mass storage ROM.
PLHOOK	103661	If the PLOTTER IS select code is other than one or two, PLHOOK gets called. The contents of R30-R31 determine what routine is executed.
PRSIDL	103733	Parser intercept. Should be taken anytime you want to alter the way something is parsed by the system or if the system can't parse something.
DEF SPAR0	104011	One of the two spare hardware hooks (currently used by the system monitor).
SPAR1	104022	Second spare hardware hook.
STRANGE	103715	Parameters for parsing functions are usually numeric, array, or string types. When the system encounters a parameter not of one of these types, it is of type strange. The STRANGE hook is called and parsing this parameter is up to the programmer.

The hooks RMIDLE, CHIDLE, and IOSP are directly accessed by the executive loop. The following code shows how to take control at these hooks.

RMIDLE

Starting at the RAM location 1Ø37Ø6, room is allowed to store the following 7 bytes of code:

```
JSB =ROMJSB      3 bytes - used to select external ROM (if
                           needed).
DEF LABEL        2 bytes - the address of the routine that will
                           be written by the programmer.
VAL ROM#         1 byte  - the number of the external ROM that
                           will be accessed using ROMJSB.
RTN              1 byte  - return to the executive loop.
```

Since ROM Ø is usually selected when the system is in the executive loop, external ROMs must go through ROMJSB in order to be selected. Binary programs need only store the following 4 bytes:

```
JSB =LABEL       3 bytes - subroutine jump to programmer's
                           routine.
RTN              1 byte  - return to the executive loop.
```

The following two pieces of code are examples of how to take control of a hook from a ROM and from a binary program.

From a ROM:

```
LDM R41,=316            Opcode for 'JSB ='
DEF ROMJSB              Address of the ROMJSB routine
DEF LABEL               Address of the hook routine
VAL ROM#                Number of the external ROM
RTN                     Return to the executive loop.
STMD R41,=RMIDLE        Store the subroutine jump to the
                        hook routine LABEL at the RMIDLE
                        location.
```

Since the DRP is set to R41 in the first instruction, seven bytes will be loaded, which will include the 316 (JSB =),the DEF ROMJSB, the DEF LABEL, the VAL ROM#, and the RTN. The code itself will not be executed until the executive loop accesses RMIDLE.

From a binary program:

```
LDM R44,=316          Opcode for "JSB ="
DEF LABEL             Address of the hook routine
RTN                   Return to the executive loop.
ADMD R45,=BINTAB      Finds the absolute address of the
                      label LABEL.
STMD R44,=RMIDLE      Store the subroutine jump to the
                      hook routine LABEL at the RMIDLE
                      location.
```

Here, the 316 opcode, the DEF LABEL, and the RTN are loaded into R44-R47. BINTAB can be safely added to the address LABEL, even though LABEL is a two-byte address and BINTAB is a three-byte address. This is because the most significant byte will be added to the RTN and the most significant byte of BINTAB is always zero and will not affect the RTN opcode. The absolute address of LABEL will always be less than 177400, the limit of binary program memory.

The normal method of returning to the system from RMIDLE is to execute a RTN instruction. Nothing will be on the R6-R7 stack except the return addresses from RMIDLE.

CHIDLE

When a key is pressed on the keyboard, the keyboard controller will generate an interrupt request which causes control to pass to the key-service routine. The key-service routine will immediately execute a reset when the [RESET] key is pressed. If no other key is being processed at the same time, the keycode is stored in the location called KEYHIT. The flags are set in XCOM and SVCWRD that indicate that the keyboard is awaiting service for its interrupt. The keyboard controller is reset, and the key service routine returns to whatever it was doing. The next time execution returns to the executive loop XCOM is checked for any pending service requests. If there are any pending requests, the executive loop checks SVCWRD to see which device needs servicing.

In this case the keyboard is the interrupt device, and the executive loop will call the character editor (CHEDIT). CHEDIT will do three things before processing the character input:

1. Set binary arithmetic mode.

2. Clear the E register.

3. Jump control to the location CHIDLE.

At this point you can check the contents of KEYHIT to determine if you want to return to the system or handle the key.

In order for a binary program to handle the key you must pop two return addresses off the R6 stack to insure returning to the executive loop and not to return to CHIDLE or CHEDIT. You must also execute a JSB =EOJ2. This routine clears the bit in SVCWRD which indicates the keyboard needs servicing, and if no other devices have requested service, clears the service request bit in R17.

The status of the E register should also be checked before returning to the executive loop. The E register is cleared by CHEDIT before calling CHIDLE and expects it to be cleared before returning back to CHEDIT. If the E register is nonzero when you return, it assumes that the key pressed was [END LINE] and tries to parse whatever is in the input buffer (INPBUF).

The following section of code illustrates how to take over CHIDLE:

```
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 RTN.
LDB R44, =316          Load the opcode for JSB.
STMD R44, =CHIDLE      Store it all (multi-byte store) to
                       CHIDLE hook.
```

IOSP

When an interface module generates an interrupt, the CPU jumps control to location IRQ20, which is usually taken by the I/O ROM. If IRQ20 has not been taken, the interrupt is ignored. The IOSP interrupt hook is accessed through the executive loop. The I/O ROM IRQ20 routine does minimal interrupt processing and sets the CSTAT and XCOM flags to indicate that an interrupt has occurred. This causes the executive loop to jump to IOSP, where the I/O ROM finishes processing the interrupt. If you take IOSP you must clear the service bit in CSTAT before returning.

Initialization Hooks

A routine called ROMINI is called on several occasions to perform initialization in external programs. Power-on, allocation, reset, deallocation, and executive loop hooks are times when the binary program may need to initialize special values. When this occurs, the initialization routines in binary programs and ROMs are given control.

A parameter is passed to the ROMINI routine through ROMFL. The occasions and corresponding ROMFL values are:

ROMFL Value	Function
Ø	Power on
1	RESET key
2	SCRATCH
3	LOADBIN
4	RUN, INIT
5	LOAD
6	STOP, PAUSE
7	CHAIN
1Ø	Allocate token with class greater than 56.
11	Deallocate token with class greater than 56.
12	Decompile token with class greater than 56.
13	Program halt because of an error.

These calls to the ROMs and binary programs allow these programs to initialize or 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 R2Ø. Similarly, no initialization routine should use CPU registers other than R34-R37 until it is verified that the value of ROMFL is not 1Ø, 11, or 12. Once this is verified, all CPU registers numbered higher than 2Ø may be used.

Error Handling

When an error is detected inside the executive loop, a system routine immediately reports the error and waits for the error to be corrected. The first 1Ø (octal) error numbers are default math errors which do not stop execution after the warning is reported. The routine which has found the error supplies a default value, and the processing continues. The defaults must be turned off in order to stop the execution.

The routine that displays the warning message, or sets the error flags if no other errors have occurred begins at location ERROR. When setting an error, the subroutine will use the next byte after the return address as the error number.

The subroutine ERROR has three basic parts to its operation:

- Initializing the error information.
- Interpreting error status.
- Carrying out the appropriate action.

ERROR saves the address that it will return to in R36-R37, increments it, and stores it on the R6-R7 stack. Then it finds the error number which is stored at the return address and puts it into R20-R21 after saving the previous contents. Checks are made to determine the proper action for the routine. If an error has already been found, then the routine restores the previous contents of the registers and returns immediately. If the error number is less than 10 (octal) or greater than 366 (octal), then the warning for the error is immediately displayed, and the contents of the registers restored before returning. If 'ON ERROR' has been declared and a program is running or if error defaults are off, then the error number, line number, and ROM number (if any) are stored, bits 6 and 7 in XCOM are set to 1, and the previous contents of the registers are restored before returning.

A subroutine jump to ERROR+ is equivalent to a subroutine jump to ERROR followed by a return.

An error condition tested by an assembly language program would go through the following steps:

1. The assembly language program finds an error and calls the system routine ERROR.

2. ERROR checks to see that no other errors have occurred which haven't been reported yet, in which case ERROR returns without doing anything (because only one error can be in process at a time). Otherwise, ERROR sets the error flags in XCOM and in other RAM locations such as ERRORS, ERLIN#, and ERNUM#.

3. Control returns to the assembly language program which returns to the system interpreter.

4. The interpreter will check the error flag in XCOM and, noting that it is set, will exit from the interpreter loop back to the main body of the executive loop.

5. The executive loop will see that XCOM is not 0 and will see that an error has occurred and will jump to the error-reporting routine REPORT.

6. REPORT checks to see if ON ERROR has been declared and a program is running. If so, it sets CSTAT to 'run in middle of line', changes the BASIC program counter to the next line and returns to the executive loop without printing the error message. If a program is not running or ON ERROR has not been declared, then REPORT prints the error message and returns to the executive loop.

7. The executive loop checks CSTAT to see if 'run in middle of line' is set. If so, control returns to the interpreter, and the program continues running. Otherwise, ROMFL is set to 13 and ROMINI is called, which is the routine that calls initialization routines in all the external ROMs and the binary programs. When ROMINI returns to the executive loop, CSTAT is set to idle mode.

3.8 Extended Memory Controller

Addresses Ø to 177777 (octal) can be directly accessed using 16-bit addressing. The extended memory controller (EMC) is used to access memory locations above 177777. Communication with the EMC, as with the CRT and keyboard controllers, is through the I/O addresses 177400 through 177777. Access to these locations above 177777 is through two pointers, PTR1 and PTR2.

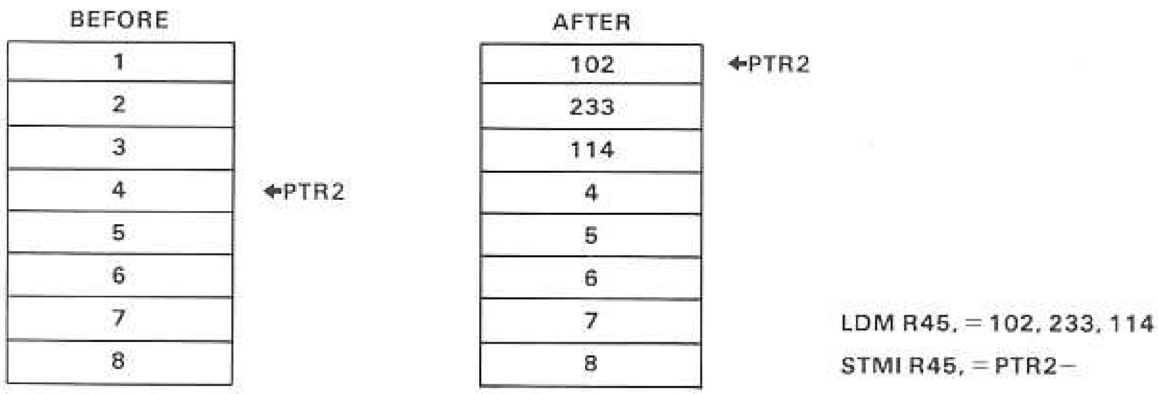
The pointers determine where in memory an access will occur, and since they must access memory locations greater than 177777, they are three-byte quantities. To set the contents of the pointers, a direct store must be performed. For example, STMD R55,=PTR2 will take the three bytes in R55-R57 and move them to PTR2 in extended memory. To store data at the desired location in memory, an indirect store must be performed. For example, STMI R32,=PTR2 will put the two bytes contained in R32, R33 at the address stored in PTR2.

The EMC pointers may be used to create stacks, with the special I/O addresses provided for each pointer. The two pointers are entirely independent of each other. Although PTR2 is used in the following examples, PTR1 and PTR2 function the same.

Each pointer has four I/O addresses: PTR1, PTR1-, PTR1+, PTR1-+, PTR2, PTR2-, PTR2-+. PTR1 and PTR2 act as pointers to memory and must be given a value in order to use the other functions. If data is stored at PTR2, it fills the memory starting at the address stored in PTR2, moving toward the higher numbered addresses.

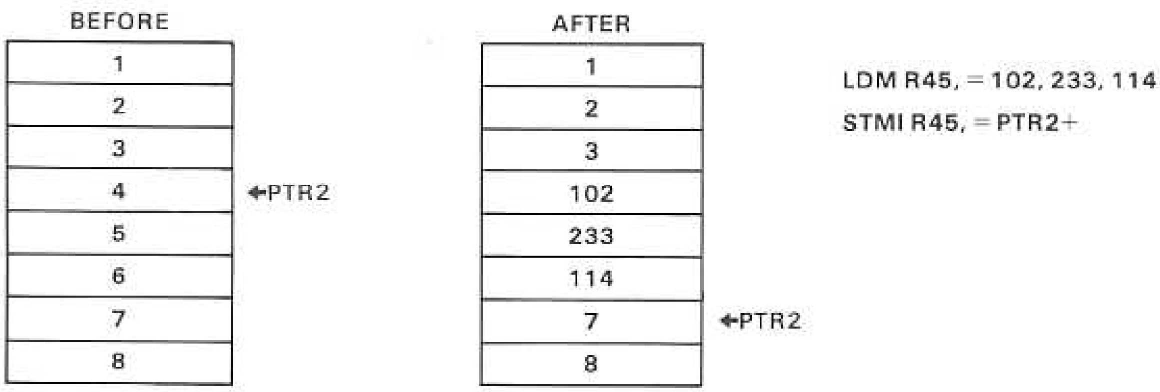
PTR2- acts as a decreasing stack pointer. A LOAD or STORE through PTR2- will first decrement the pointer by the appropriate number of bytes. The LOAD or STORE operation will then be performed, leaving the pointer at the new location.

LDM R45,=102,233,114 STMI R45,=PTR2-



PTR2+ is an increasing stack pointer which will perform the load or store operation at the location pointed to by the pointer, and then will increment the pointer after the load or store operation by the appropriate number of bytes.

LDM R45,=102,233,114 STMI R45,=PTR2+



When the CPU accesses an I/O address directly, it causes the controller to respond to the address. Each of the controllers is linked to the bus and monitors the information that is being passed from memory to the CPU. For example, the direct access instruction LDBD R32,=CRTDAT will fetch an address from memory. If this address is one which the controller must use for an operation, the controller will send an information byte to the CPU to tell it what to do. In this case the CRT controller will send the CPU the current status of the CRT.

The EMC must constantly monitor the machine code instructions being fetched by the CPU, since the DRP setting determines how many bytes are to be used in a given operation. Whenever a DRP instruction appears, it must store that information to keep track of the current DRP setting.

This can be done with PAD (restore status) and SAD (save status) instructions. SAD pushes three bytes onto the R6 stack containing information about the ARP, the DRP, and the status flags. PAD restores this information using these bytes.

Because of this method of keeping track of the DRP setting, there are cases where the EMC cannot know the DRP setting which include:

- After a PAD instruction: Since the PAD instruction restores status and the ARP and DRP settings, the EMC is not aware of what the DRP setting is until another DRP instruction is executed. Therefore you should avoid using the following or similar code:

PAD	Restores status, the ARP, and DRP
LDMI R#,=PTR2	Fetches bytes from extended memory. The CPU assumes the number of bytes is determined by the PADs DRP. whereas the EMC is using the last DRP instruction.

- When the DRP is set indirectly by the contents of CPU register RØ, as in the following case:

LDMI R*,=PTR2-	This sets the DRP according to the least significant six bits of RØ, which the EMC knows nothing about.

Because of the first situation, all interrupt service routines must be written to save and restore the contents of the registers used before returning to the routine that was interrupted. Interrupt service routines are those that are called immediately when a hardware interrupt occurs, such as a key being pressed or an I/O module needing attention. Because the interrupt is usually granted almost immediately by the CPU, interrupts can occur between any two instructions (as long as interrupts are enabled). Before restoring everything, you must do the following to solve this problem:

- Pop the SAD status information off of the R6 stack to get a copy.
- Push a copy back on for the eventual PAD.
- Figure out what the actual DRP needs to be.
- Put the appropriate DRP instruction into RAM along with a RTN.
- Restore all the registers and status (PAD).
- Jump to the DRP and RTN instruction so the EMC will get its DRP pointer back to the right value.

There is another I/O address that the EMC listens to. If you store a 1 to RULITE (177704), the power light will start blinking. If you store a 0 to that address, the light will stop blinking. This light blinks when a BASIC program is running, or when an HP-85 BASIC program is being translated, or when a program is temporarily halted waiting for input. It normally stops blinking when program execution is complete, if program execution is halted by an error, or if the program is paused.

3.9 Parsing

When you type in a BASIC program as a series of ASCII characters it is translated (parsed) and stored internally as a stream of tokens and associated data and addresses. The tokens represent the BASIC reserved words, functions, operators, and punctuation. The data bytes represent the constants, variables, and line number references.

Parsing begins with the line number or the first character of the statement and moves to the right, processing each character and space. Multiple nonquoted spaces are ignored during parsing except those occurring at the beginning of a program line. As a line is parsed, it is checked for syntax errors, changed to RPN (Reverse Polish Notation), and converted into tokens which are stored internally.

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 can be found in section 8. ASCII codes can be found in the HP-87 owner's manual.

Example: In parsing the line

```
10 LET A = B * SIN (45),
```

the system produces the following tokens in the order shown.

Tokens (Octal Value)	Comments
16	End of statement.
10	Store numeric value token.
52	Multiply token.
330	Sine token.
105	BCD 45 in integer format.
0	(Refer to paragraph 3.9, Numeric
0	Formats.)
32	Integer constant token.
102	ASCII "B", variable name.
1	Length of variable name.
0	Variable address space for allocation.
0	(Refer to Format of BASIC Programs and
0	Variables, paragraph 3.12.)
1	Fetch simple numeric variable
101	ASCII "A"
1	Length.
0	Variable Address Space
0	
0	
21	Store simple numeric variable token.
142	Let token.
25	Length of line in bytes.
20	Line number in BCD (two digits per
0	byte except for most significant
0	byte which contains only one).

The extended memory pointer, PTR2, is used as the output pointer during parsing. Tokens are stored indirectly to PTR2-. At the beginning of the parsing process PTR2 is set equal to NXTMEM, so the parsed line will be built up in available memory at the end of the last BASIC program. 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 place holder byte; STSIZE is a pointer to the place holder byte. In order to find a match for the keyword LET, the system looks first in keyword tables in the resident binary programs, then in any external ROMs, and finally in the internal system keyword table. For this reason, a binary program or external ROM can take over any keyword (that is, a binary program can implement a custom version of PRINT, while the preprogrammed PRINT is ignored). The extend register indicates if the token searched for has been found. Refer to the section on status indicators in paragraph 2.3.

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 at the end of the BASIC program and is executed immediately, being discarded when execution is finished.

For further details of parsing operations and register conventions at parse time, along with specific parse routines, refer to the system routines which are listed in alphabetical order in section 8.

3.10 Decompiling

Programs or statements are decompiled as they are listed. This is the reverse process of parsing and compiling. Internally, it requires the reconstruction of code as it was entered. The tokens which have been parsed into RPN and distributed in the system are reassembled.

PTR1 points to the input stream, which is accessed by loading indirect through PTR1-. Input is then decompiled to an expression stack or an output stack. The expression stack (R12) is used to reconstruct expressions from RPN to their original form, and an output stack (pointed to by R30) is used to buffer the output.

Since the tokens are arranged in RPN internally, the system decompiles the tokens as it pushes missing operator tokens (016) onto the expression stack. These missing operator tokens are merely "place holders" 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.

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. Following 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, such as, A(3)	() to expression stack if token odd; otherwise (,) to expression stack.
34	Dimension subscript like, 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.
5Ø	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 5Ø.
53	String binary operator	Same as class 51.
55	Numeric function	Replace the most recent missing operator with "," for each parameter. Then insert function name (at most recent missing operator) and push onto expression stack.
56	String system function	Same as class 55.

The following example illustrates how decompiling occurs:

10 LET A=B*SIN(45)

After being parsed as shown, these tokens are decompiled into the output stack and the expression stack as illustrated.

STEP	TOKENS	R12 EXPRESSION STACK	OUTPUT BUFFER
9.	16 EOL	———	10 LET A = B * SIN(45)
8.	10 =	16 A = B * SIN(45)	10 LET
7.	52 *	16 A 16 B * SIN(45)	10 LET
6.	330 SIN	16 A 16 B 16 SIN(45)	10 LET
5.	105 0 0 32 } 45	16 A 16 B 16 45	10 LET
4.	102 1 0 0 0 1 } B	16 A 16 B	
3.	101 1 0 0 0 21 } A	16 A	
2.	142 LET	———	10 LET
1.	25 LEN 20 0 0 } LINE #	———	10

3.11 Operating Stack

The stack to which R12 points is used for passing values in many internal system routines. The formats of values that are fetched and stored during run time execution of certain specific tokens, as well as the formats of numeric quantities are in this section.

Numeric Formats

In internal routines, numbers popped off the R12 stack are eight bytes long, so integer values are tagged with octal 377.

	Real		Tagged Integer		
R40	E1	E2			Undefined
R41	E0	MS			Undefined
R42	M10	M11			Undefined
R43	M8	M9			Undefined
R44	M6	M7	377		← Integer Tag
R45	M4	M5	D1	D0	
R46	M2	M3	D3	D2	
R47	M0	M1	S	D4	

NUMERIC FORMATS (R12 STACK)

In the illustration, 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. The numbers are shown as they would be if they were taken off of the stack by the instruction POMD R4Ø,-R12. The tagged integer is right justified so that the most significant digits (starting with D4) are Ø if unused. For tagged integers, the decimal point is to the right of DØ, the least significant digit. The real number decimal point is between MØ and M1.

A short numeric variable is formatted as follows:

R44	E0	E1
R45	M3	M4
R46	M1	M2
R47	0 0 SM SE	M0

EØ	Most significant four-bit BCD digit of the exponent.
E1	Least significant four-bit BCD digit of the exponent.
MØ	Most significant four-bit BCD digit of the mantissa.
M4	Least significant four-bit BCD digit of the mantissa.
SM	Sign of the mantissa (Ø=positive, 1=negative).
SE	Sign of the exponent.

The decimal point is assumed to be between digit MØ and digit M1. The most significant nibble (four bits) contains the signs of the mantissa and the exponent. The two most significant bits are zeroes.

Strings on the R12 Stack

String values are passed on the operating stack as a two-byte length and a three-byte address of the next character higher than the first character of the string. The first character is at the highest address of any characters of the string. To fetch successive characters of the string, the following code could be used:

```
        POMD R45,-R12         ! Get the address of $
        STMD R45,=PTR2-       ! Set PTR2 pointing to first character
        POMD R36,-R12         ! Get the length of $
LOOP    LDBI R32,=PTR2-       ! Get the next character
             .
             .
             .
        DCM R36               ! Decrement length count
        JNZ LOOP              ! Loop until done
```

Operating Stack Routines

There are several system routines available to help you in parsing various kinds of parameters for BASIC statements. These routines will parse your BASIC statement into tokens that, at run time, will load the R12 stack with the appropriate variable or parameter.

Following is a list of the routines that can be used and what they leave on the stack:

NUMCON	(8 bytes) Real or tagged integer.
NUMVAL	(8 bytes) Real or tagged integer.
REFNUM	(3 bytes) Absolute address of variable value. (3 bytes) Absolute address of name of variable. (1 byte) Head of variable.
STRCON	(2 bytes) Length of string. (3 bytes) Absolute address of string.
STREXP	(2 bytes) Length of string. (3 bytes) Absolute address of string.
STRREF	Will parse both a normal string variable and a string array variable reference. There will be slightly different information on the stack depending on which of these it is. String arrays will have everything that nonarray strings will have but string arrays may also have row, column, and dimension information if the variable is being traced. You can tell if that information is there by checking the trace bit in the header byte which will come off the stack before the tracing information would. You also tell whether you have a string array or normal string by inspecting the appropriate bit in the header byte.

Nonarray Strings

(3 bytes)	Absolute address of name of variable.
(1 byte)	Header of variable.
(2 bytes)	Maximum length of string variable.
(3 bytes)	Absolute address of first byte of string address.
(2 bytes)	Maximum length available to store into. This will be different from the maximum length if subscripts were used.
(3 bytes)	Absolute address of first byte to store into. This will also be different from the address of the first byte of the string if subscripts were used.

Array Strings

The first three will only be on the stack if the variable is being traced.

(2 bytes) Row of element.
(2 bytes) Column of element.
(1 byte) Dimension flag (Ø=2 dim., 1=1 dim.).

(3 bytes) Absolute address of name of variable.
(1 byte) Header of variable.
(2 bytes) Maximum length of string variable.
(3 bytes) Absolute address of first byte of string variable.
(2 bytes) Maximum length available to store into. Different than maximum length of variable if subscripts were used.
(3 bytes) Absolute address of first byte to store into. Different from the address of the first byte if subscripts were used.

NARREF Used when you wish to use a simple numeric variable name to refer to an array variable. An example would be:

```
MAT C=ZER
```

In this example 'C' refers to an array C, not to a simple numeric variable.

(3 bytes) Address of variable header. This address is a relative address. The easiest way to make it an absolute address is:

```
POMD R65,-R12
JSB =FETSVA
```

FORMAR Used when you wish to refer to an entire array.

```
PRINT# 1; C(),D(,)
```

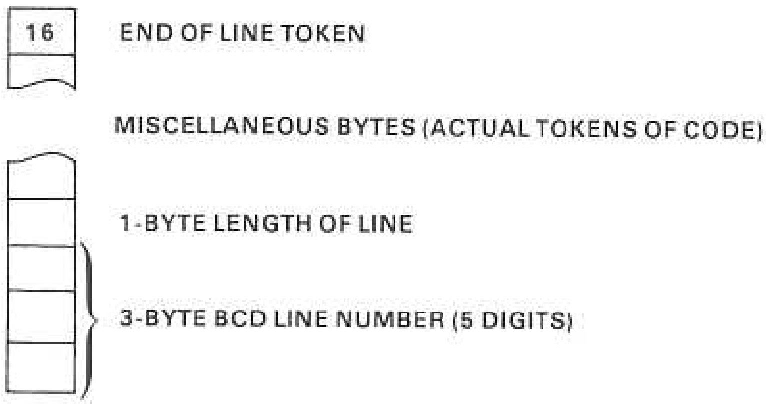
(3 bytes) Absolute address of the first element of the array.

(3 bytes) Absolute address of the array name.
(1 byte) Array header.

In all of the above examples of stack contents, the bottom of the page represents the direction of higher addresses. As you popped things off the stack you would be removing things from the bottom first.

3.12 Format of BASIC Programs and Variables

The following figure shows how a BASIC program line is formatted:



The BASIC line

```
15160 END
```

would be parsed as:

```
Ø16 'END OF LINE' TOKEN
212 'END' TOKEN
ØØ2 LENGTH OF LINE (212 AND Ø16 MAKES TWO BYTES)
14Ø
121 --> 3-BYTE BCD LINE #
ØØ1
```

Let's take a look at how a line number of 1516Ø generates the three bytes 14Ø, 121, and ØØ1. Since a BCD digit takes four bits, two digits can be packed into one byte. So, let's split the line number into three digit groups:

```
1    51    6Ø
```

Now we turn those groups into bits:

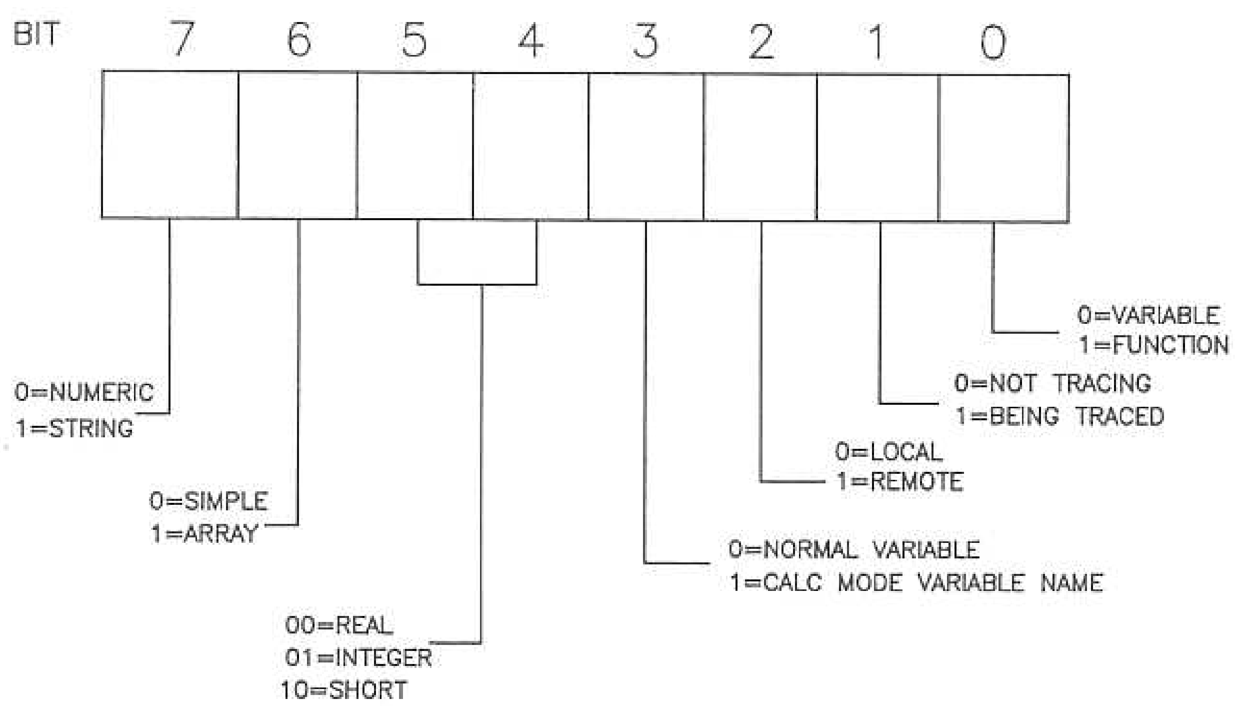
```
  1     ØØØØ ØØØ1
5 1     Ø1Ø1 ØØØ1
6 Ø     Ø11Ø ØØØØ
```

Arrange the binary representation with three to a group. Convert this form to an octal number to obtain the three bytes that represent the line number.

00 000 001	001
01 010 001	121
01 100 000	140

The values of the variables are stored at the end of the current program in one continuous block of memory. Each variable has a header which contains information about that variable. Following are the structures of different kinds of variable storage areas. All variable storage areas begin with a one byte header. The bits in that header and their meanings are:

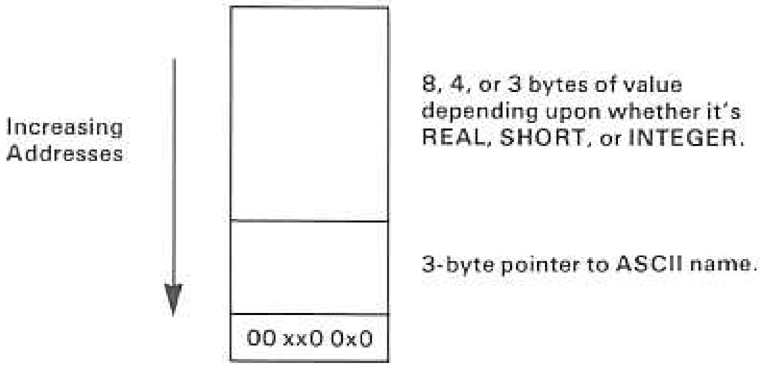
VARIABLE HEADER BYTE LEGEND



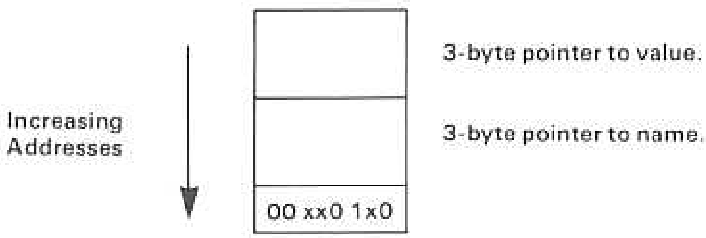
In the following diagrams in this section, an "x" will mean that that particular bit position can be occupied by a "1" or a "Ø."

Simple Numeric Variable

Local

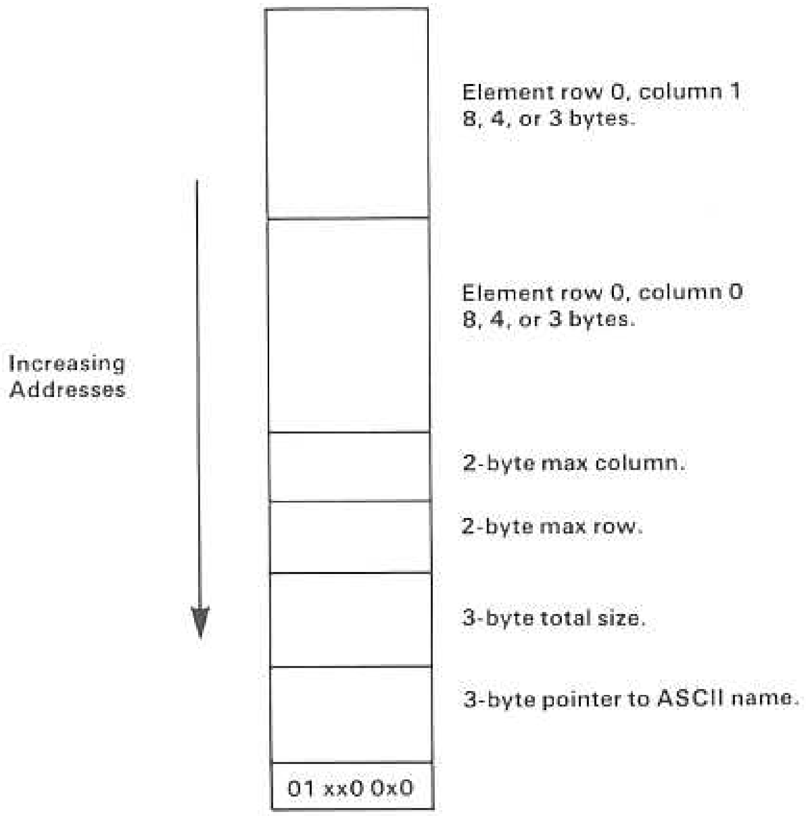


Remote

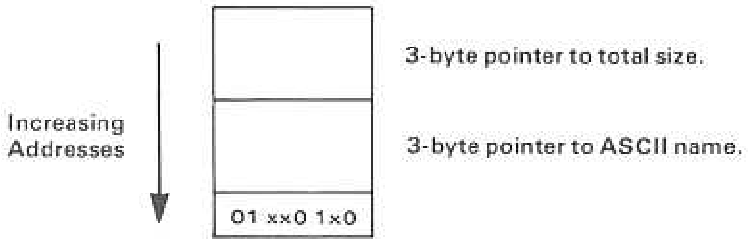


Numeric Array

Local

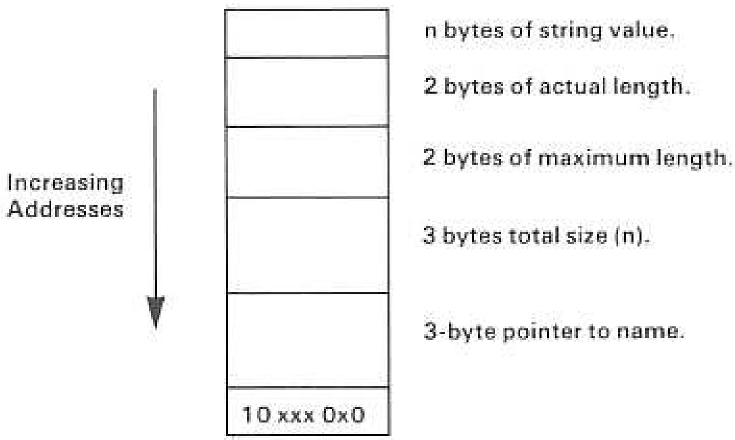


Remote

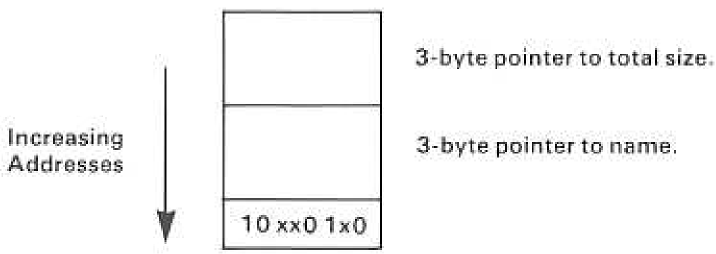


Simple String Variable

Local

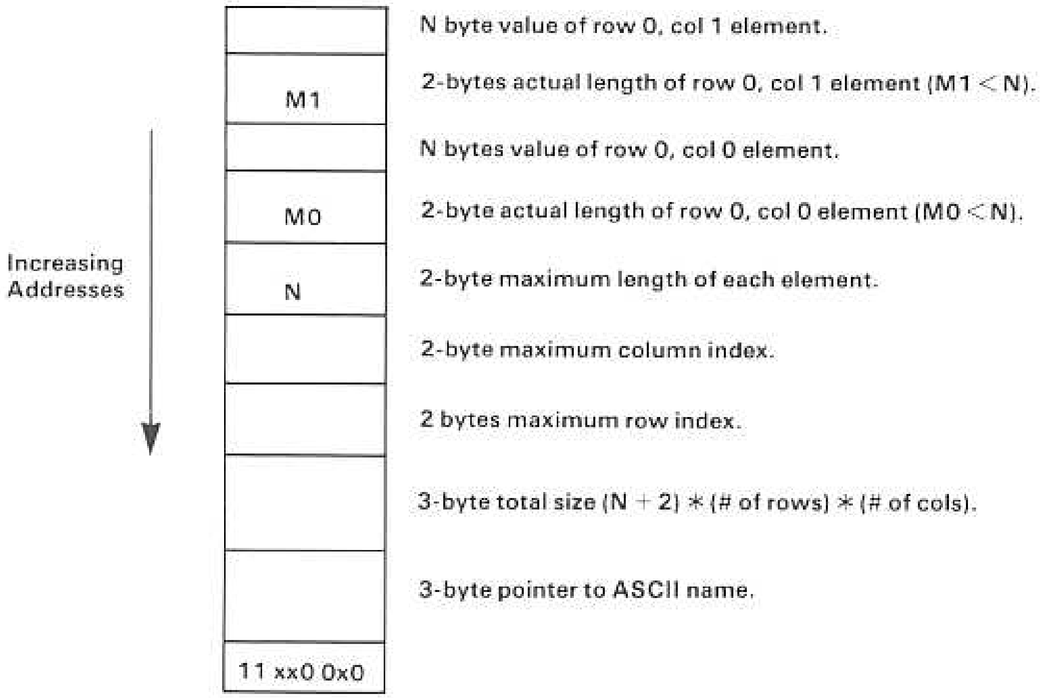


Remote

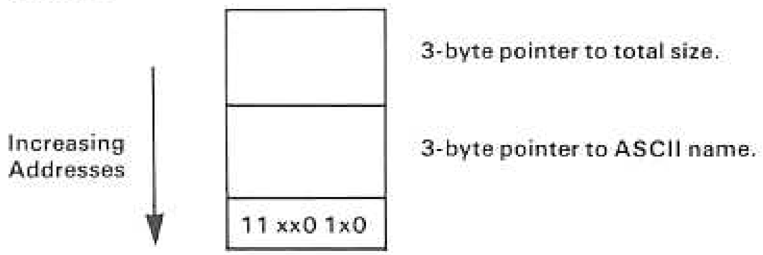


String Array Variable

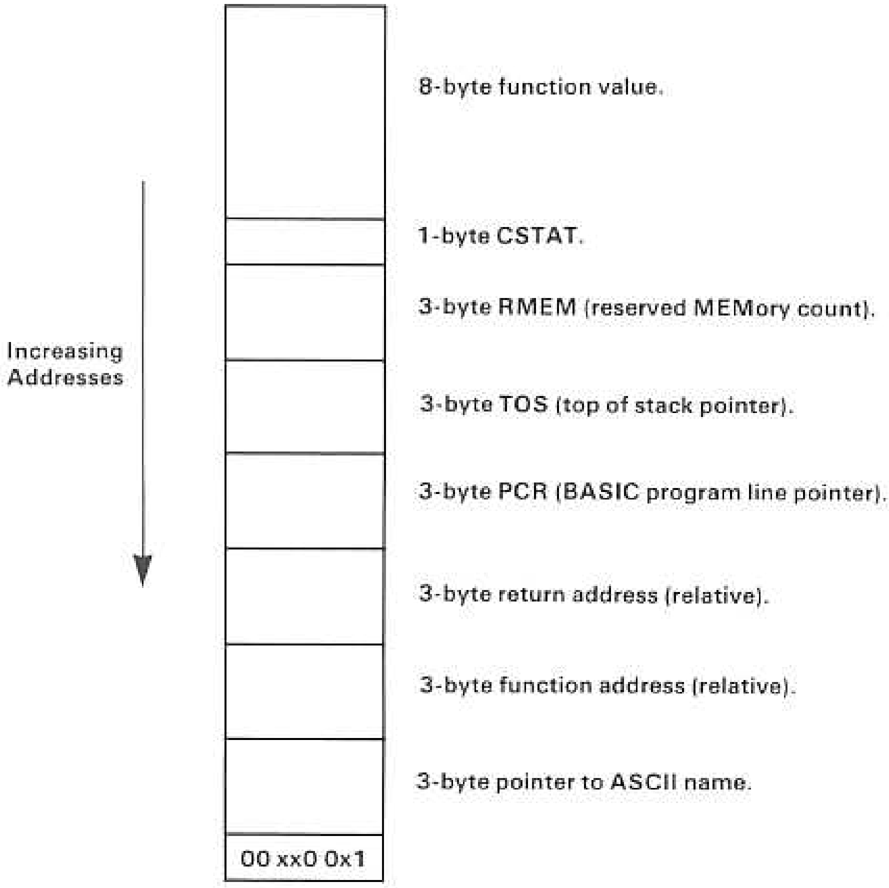
Local



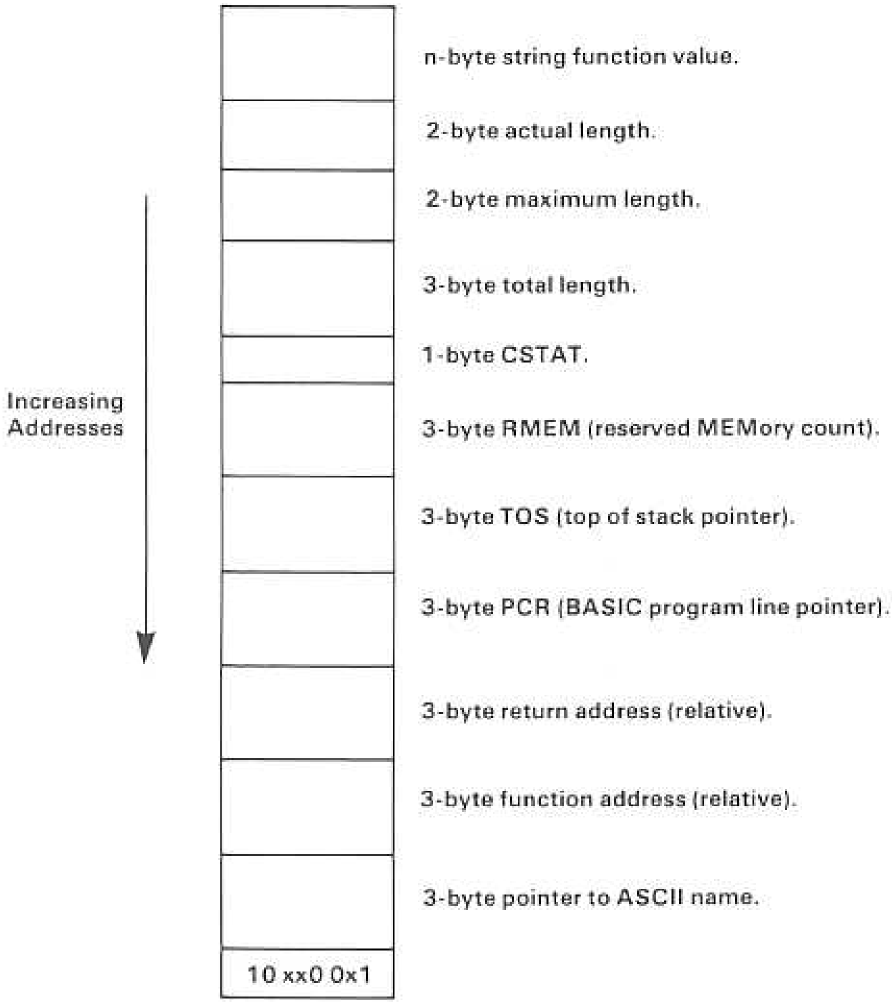
Remote



Numerical User Defined Functions



String User Defined Functions



Because calculator mode statements destroy all previous calculator mode statements but not their variables, the pointers to the ASCII names of the variables cannot point to the calculator mode statement. A dummy calculator mode simple string variable is created with the bit set in the header that indicates this is a calculator mode variable name. This dummy variable is skipped for all purposes other than searching for variable names at allocation time for calculator mode statements. When a calculator mode statement is allocated, the addresses used for the variables are relative to FWCURR.

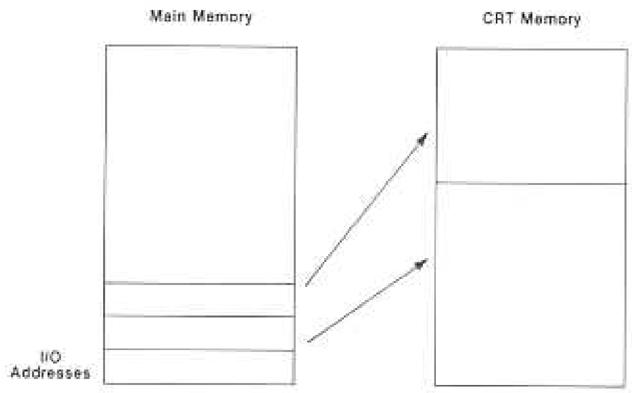
CONTROLLERS

4.1 Introduction

The HP-87 is a multi-processor system. The keyboard, the CRT, the timers, and the interface modules are all controlled by individual microcomputers. The mainframe CPU coordinates activities between the peripherals using the I/O addresses. To communicate with these controllers, refer to the appropriate sections.

4.2 CRT Controller

The CRT is an intelligent component that is controlled by an internal computer, or CRT controller. The CRT also has a memory which continuously refreshes the CRT display.



The CRT controller and the CPU communicate using four addresses in RAM. Each address requires a two-byte quantity to specify a CRT memory address. The I/O addresses are:

CRTBAD DAD 177701
Storing a two-byte address to this location causes the CRT controller to load its byte address pointer with that address.

CRTSAD DAD 177700
Storing a two-byte address to this location causes the display to be started at that address. This makes the display appear to scroll up and down or side to side or to jump to a different page depending on the new start address. Storing to CRTSAD has no effect when in GRAPH NORMAL or GRAPH ALL modes.

CRTDAT DAD 177703
Storing a single byte to this location causes that byte to be stored to the CRT memory location currently pointed to by the controllers byte address. Loading a single byte from this location reads the byte from the CRT memory location currently pointed to by the controller's byte address.

After either a load or store operation through CRTDAT the CRT controller automatically increments by one its internal byte address pointer. If you did a series of single byte store instructions to CRTDAT without storing anything to CRTBAD in between, those bytes would be stored in successive CRT memory locations.

However, before storing to CRTDAT, you must first read CRTSTS and check the least significant bit to make sure the controller is not busy. Before loading from CRTDAT, you must store a byte to CRTSTS with the least significant bit set to tell the CRT controller that you want to read the current memory location. You must then read CRTSTS until the BUSY bit indicates the controller is not busy, at which point you can load from CRTDAT to get the byte. An easier way is to simply execute a JSB =INCHR (call the system routine) that does all the rest for you.

CRTSTS DAD 177702
Loading a single byte from CRTSTS gets you information about the current status of the CRT controller. Each bit has a specific meaning:

Reading from CRTSTS

Bit	0	1
0	Not Busy	Busy
1	Unblank	Blank
2	Power-up	Power-down
3	16 lines	24 lines
4	Display time	Retrace time
5	Noninverse	Inverse Display
6	Normal	All
7	Alpha	Graphics

Storing a single byte to CRTSTS sets the CRT controller to a specific mode and/or requests a read:

Storing to CRTSTS

Bit	Ø	1
Ø	No read	Read Request
1	Unblank	Blank
2	Power-up	Power-down
3	16 lines	24 lines
4	-	-
5	Noninverse	Inverse Display
6	Normal	All
7	Alpha	Graphics

When the CRT is blank, the controller has disabled the electron beam, causing the display to go blank. When this is the case, the controller does not have to refresh the display, causing it to transfer data to and from the CRT memory much faster. When you switch from alpha to graphics or graphics to alpha there will be a flash on the display unless it has been blanked. To avoid this, you must set the blank bit during a retrace. There is a pair of system routines that will blank and unblank the CRT for you. They are CRTWPO and CRTUNW.

When the CRT is powered up or powered down, the controller turns the high voltage section of the CRT driver on or off. This is done to conserve power.

4.3 Display Modes

ALPHA NORMAL

OCTAL ADDRESS	
000000	
000120	
000240	
007760	
010100	
010220	last line of ALPHA NORMAL memory
010340	start of GRAPH NORMAL memory

ALPHA addresses in CRT memory are ØØØØØØ to Ø1Ø337. In alpha mode the display shows 16 or 24 (decimal) lines of 8Ø (decimal) characters per line. The scrolling keys permit viewing of an additional 38 (decimal) lines of alphanumeric data.

Because each ASCII character occupies eight bits, one character can be stored at each memory location. To move the cursor to the right one position, add one to the address.

If the start address (CRTSAD) is at an address where there is not enough ALPHA NORMAL memory left for an entire display, then the CRT controller will start fetching bytes from address ØØØØØØ when it reaches the end of the ALPHA memory. Because of this a mod operation must be performed on alpha addresses when moving the cursor around.

At power-on and after a RESET the CRT start address is set to ØØØØØ. If you roll the display up one line the CRT start address will then be set to ØØ12Ø. If you were to roll the display down one line, the start address would be 1Ø22Ø.

GRAPH NORMAL

OCTAL ADDRESS	
010340	first (top) line of GRAPH NORMAL display
010422	
037534	
037616	last (bottom) line of GRAPH NORMAL display
037700	last 64 bytes are unused.

In GRAPH NORMAL mode, the screen is 50 bytes (decimal) wide. The GRAPH display always starts at 10340.

The last 64 (decimal) bytes of CRT memory are unused in NORMAL mode. The contents of memory location 10340 will determine whether or not the first eight dots in the top line of the display will be on. The contents of memory location 10341 will determine the state of the next eight dots on the top line.

ALPHA ALL

OCTAL ADDRESS	
000000	first line of ALPHA ALL memory
000120	
037440	
037560	last line of ALPHA ALL memory
037700	last 65 bytes are unused.

The ALPHA ALL memory maps 80 addresses per line of the CRT display and the last 64 (decimal) bytes of memory are unused. When the start address gets too close to the end of memory, the controller wraps around to address 000000 to finish the display page.

GRAPH ALL

OCTAL ADDRESS	
010340	first (top) line of GRAPH ALL display
010444	
037640	
037744	addresses 37744-37777 addresses 0-47
000050	
000154	
010134	last line of GRAPH ALL display
010240	last 64 bytes are unused

In GRAPH ALL mode there are 68 (104 octal) bytes per line of the graphics display, giving a dot resolution of 544 dots wide by 240 dots high. The controller will again wrap back to address 000000 to continue fetching bytes when it runs out of memory at the end of the NORMAL graphics area.

4.4 Keyboard Controller

The keyboard controller monitors the RAM location keyboard scanner, four timers, and the beeper.

Keyboard Scanner

All of the keys are connected to keyboard inputs. The controller monitors these connections, waiting for a key to be pressed. When a key is pressed, the controller generates a service request to the CPU. When the request is granted execution vectors to the service routine KEYSRV. The keyboard service routine saves the CPU status then does a JSB=KYIDLE instruction (refer to Hooks, paragraph 3.5). If the KYIDLE hook has not been taken, control will return to KEYSRV. It will then disable interrupts, save registers, and read the key code of the key that was pressed from the keyboard controller through the I/O address KEYCOD. The key is checked by KEYSRV to see if it was RESET. If so, KEYSRV does a RESET. If not, it checks to see if any other keys have been pressed that have not been handled by the system.

If another key has been pressed, the system re-enables the keyboard scanner and restores the registers and status. The system returns to what it was doing when the CPU received the service request. As long as other keys are not pending, the key code is saved in a RAM location called KEYHIT and bits are set in R17 and SVCWRD, indicating that a key has been pressed. The routine KEYSRV then restores the registers and status.

Once a key has been pressed, no more keyboard interruptions will be seen until the previous key is released, and a 1 has been stored to I/O address KEYCOD (which restored the keyboard scanner). If the interrupt were to occur between the last DRP instruction and an extended memory access, the EMC could lose track of what the DRP setting is. Refer to paragraph 3.6.

The following define the I/O addresses associated with the keyboard scanner:

KEYSTS

Write: Bit	Ø	1
Ø	No effect	Enable keyboard
1	No effect	Disable keyboard
2	Not used	-
3	Not used	-
4	not used	-
5	Speaker off	Speaker on
6	No effect	1.2 kHz
7	No effect	Toggle Flip FF
Read: Bit	**Ø**	**1**
Ø	Device disabled	Device enabled
1	No key pressed	Key pressed
2	Not used	-
3	Shift key up	Shift key down
4	Not used	-
5	Not used	-
6	Not used	-
7	Globals disabled	Globals enabled

Bits Ø and 1 of KEYSTS allow you to disable and enable the keyboard separately from all other devices. Bit 7 (when reading) tells you whether global interrupts are enabled or disabled.

KEYCOD

The status of KEYCOD utilizes a byte rather than individual bits.

Write: If the value is 1, then the keyboard scanner will be re-enabled as soon as the key is released.

Read: Returns the keycode of the key that was pressed.

Following is a listing of the system key service routine, KEYSRV, presented here as an example of what you need to do if you take over KYIDLE.

```
KEYSRV   SAD                     ! Save the STATUS, ARP, and DRP
         JSB =KYIDLE             ! Call the RAM hook
         STBD R32,=GINTDS        ! Disable global interrupts
         PUMD R32,+R6            ! Save register contents to recall later
         LDBD R32,=KEYCOD        ! Get the keycode from the controller IC
         BIN                     ! Force BIN mode for keycode compare
         CMB R32,=213            ! Is it the RESET key?
         JNZ NORSET              ! Jif no
RSTART   LDM R6,=STACK           ! Else reset the return stack pointer
         JSB =RESET.             ! Do a RESET
         LDB R30,=1              ! Need to store a 1 out to KEYCOD to
         STBD R30,=KEYCOD        !      restart the keyboard scanner
         GTO DOCUR.              ! Output cursor to CRT and fall into exec.
NORSET   LDBD R33,=SVCWRD        ! Any other unserviced keys been pressed?
         JOD HAVE1               ! Jif yes, throw this key away
         ICB R33                 ! Else set the keyboard bit
         STBD R33,=SVCWRD        ! And restore SVCWRD
         STBD R32,=KEYHIT        ! Save the keycode for the system
HAVE1    LDB R32,=20             ! Load the mask to set service request bit
         ORB R17,R32             !       in XCOM (R17)
         JSB =EOJ1               ! Make sure we're set to slow repeat speed
         LDB R#,=1               ! Can't get any more keys unless we
         STBD R#,=KEYCOD         !       restart the keyboard scanner
         POMD R#,-R6             ! Restore the registers we used (R32-R33)
         GTO ENDSR               ! Make the current DRP setting available
!                                  to the EMC.
!
EOJ1     LDBD R32,=KRPET1        ! Get the slow repeat count
         STBD R32,=KEYCNT        ! Set the counter to the slow repeat
         RTN
!
!
ENDSR    STMD R10,=S10           ! Save R10-11 in a reserved RAM location
         POMD R10,-R6            ! Get the byte of SAD that contains DRP
         PUMD R10,+R6            ! Restore so we can PAD later
         ANM R10,=77,0           ! Isolate the DRP register bits
         ADB R10,=100            ! Make it a DRP instruction
         STBD R10,=RAID+1        ! Store it into RAM so we can execute it
         LDMD R10,=S10           ! Restore R10-11
         GTO RAID+1              ! Finish
```

At power-on, the system initialization routine has stored at RAID+1 the following code:

```
RAID+1  BSZ 1               Place holder for DRP instruction
        STBD R#,=GINTEN     Re-enable global interrupts
        PAD                 Restore status, the ARP, and the DRP
        RTN                 Done
```

Timers

The timer section of the keyboard controller consists of four separate timers and four registers each containing eight BCD digits. The timers and registers are updated at a rate of 1 kHz. During this updating, no read or write operations should be performed to the CLKDAT address. Each timer that equals its register count causes a service request interrupt. It is then set to zero to begin another count sequence. The contents of the timers are transferred in four consecutive bytes each containing two BCD digits.

The keyboard scanner has the highest priority on the controller regarding interrupts. Next highest is timer Ø, with timer 3 being the lowest.

CLKSTS

This address contains the following information needed when using the timers.

Write: Bit	Comments
0	Disable addressed timer.
1	Enable addressed timer.
2	Stop addressed timer.
3	Start addressed timer.
4	Clear addressed timer.
5	Clear interrupt service flip FF.
6	Bits 6 and 7 are the
7	timer address (0 through 3).
Read: Bit	**Comments**
0	Timer 0 enabled.
1	Timer 1 enabled.
2	Timer 2 enabled.
3	Timer 3 enabled.
4	Not used.
5	Not used.
6	Not used.
7	Read (timers available for access through CLKDAT).

CLKDAT

When loading from CLKDAT, you must execute a four-byte load to get eight BCD digits which represent the value of the last addressed timer.

When storing to CLKDAT, you must execute a four-byte store and the four bytes must be the eight-digit value you want the last addressed timer set to.

Before executing a load or store instruction to CLKDAT you must first check the most significant bit of CLKSTS to make sure the timers are ready to be accessed (bit 7=1).

There are no hooks in the timer interrupt routines. The only way to make use of the timers from assembly language programming is to periodically check SVCWRD to see if any timers have been interrupted. This will only work if you never return to the BASIC interpreter, as the executive loop will also check for timer interrupts at the end of each BASIC statement and handle them if necessary.

The following code will read the value of timer Ø (the system clock). It will use that value and the base time to generate the current time and return the current time to the R12 stack.

```
TIME.     CLB R55                  ! ADDRESS TIMER Ø
          STBD R55,=GINTDS         ! DISABLE INTERRUPTS
          JSB =TIMWST              ! WAIT FOR READY AND STORE
                                   ! TIMER ADDRESS
          CLM R4Ø                  ! CLEAR UPPER FOUR BYTES
          JSB =TIMRDY              ! WAIT FOR READY
          LDMD R44,=CLKDAT         ! TIME TO R44-R47
          STBD R44,=GINTEN         ! RE-ENABLE INTERRUPTS
          LDM R36,=4,Ø             ! SET EXPONENT
          BCD
          CLB R32                  ! SET SIGN TO POSITIVE
          JSB =SHRONF              ! SHIFT, PACK AND PUSH
                                   ! ON R12 STACK
          LDMD R5Ø,=TIME           ! GET BASE TIME
          POMD R4Ø,-R12            ! RECOVER INITIAL TIME
          JSB =ADD1Ø               ! ADD TO BASE TIME AND
                                   ! PUSH ON R12 STACK
          RTN

TIMWST    JSB =TIMRDY              ! WAIT FOR READY
          STBD R55,=CLKSTS         ! STORE OUT STATUS BYTE
          RTN

TIMRDY    LDBD R37,=CLKSTS         ! GET TIMER STATUS
          JPS TIMRDY               ! JIF BUSY
          RTN                      ! ELSE RETURN
```

The system routine SHRONF takes a 16-digit number in R4Ø-R47, an exponent in R36-R37 and a sign byte in R32 and normalizes it (shifts out leading zeroes and adjusts the exponent to match). It then packs the exponent and sign into R4Ø-R41, and pushes the floating point result onto the R12 stack. The ADD1Ø routine is basically the same as ADDROI except it expects as inputs two real (floating point) numbers in R4Ø-R47 and R5Ø-R57, rather than two real or integer numbers on the R12 stack.

The following code sets timer Ø (the system time clock) the way it's set at power-on.

```
    TIMEØ     LDB R55,=32               ! SET UP STATUS BYTE.
                                        ! BITS 4, 3, 1 WILL CLEAR
                                        ! TIMER O.
                                        ! START IT, AND ENABLE IT TO
                                        ! INTERRUPT.
              CLM R44                   ! GENERATE 864ØØØØØ, THE
                                        ! NUMBER OF MILLISECONDS
              LDM R46,=1ØØ,2Ø6          ! IN A DAY.
              STBD R#,=GINTDS           ! DISABLE INTERRUPTS.
              JSB =TIMRDY               ! WAIT FOR READY.
              STMD R44,=CLKDAT          ! SEND THE TERMINAL COUNT.
              STBD R44,=GINTEN          ! RE-ENABLE GLOBAL INTERRUPTS.
              RTN                       ! DONE.
```

Speaker

The speaker can be controlled through the I/O address KEYSTS. Bits 5 and 6 of KEYSTS allow you to either make the speaker beep at 1.2 kHz or turn it off and on at whatever frequency you wish (within the limits of the clock cycle of the CPU).

SYSTEM MONITOR

5.1 Introduction

The HP 82928A System Monitor is an optional plug-in module that permits you to set breakpoints and single step or trace through the execution of assembly code. Two breakpoints can be set in any portion of memory with an address lower than 200000. Any time either of these addresses is referenced in any manner, an interrupt is caused. The user can use this interrupt to examine CPU registers, status bits, memory locations, and extended memory pointers.

5.2 System Monitor Commands

The system monitor commands described in this section are demonstrated later in this manual. Refer to section 7.

BKP octal address [,select code for output]

Sets breakpoint (BKP) #1 or #2 at a specified address in memory. If no breakpoints have been set, the command sets BKP#1. If BKP#1 is already set, the command sets BKP#2. If BKP#1 and BKP#2 are both set, the command resets BKP#2; BKP#1 remains set at its original address. Breakpoints can be set at any address lower than 200000 in system RAM or ROM. They can be cleared only by using the CLR command. Using the [RESET] key will not clear breakpoints.

When a breakpoint is encountered, execution halts and a block of status information is output to the CRT IS device. The following keys are typing aids:

Key	Use
B	Set an additional breakpoint (BKP)
C	Clear (CLR) a breakpoint.
M	Obtain a memory dump (MEM).
P	Change program counter (PC=).
R	Change contents of a register (REG).
T	Using the TRACE command.
1	Change value of pointer #1 (PTR1=).
2	Change value of pointer #2 (PTR2=).
[STEP]	Single step execution.
[ROLL ^]	Roll up display.
[ROLL V]	Roll down display.
[RUN]	Resumes program execution.
[BACK SPACE]	Back space.
[A/G]	Alternates between graphics and alpha modes.

Most other keys on the keyboard are inactive at a breakpoint until a typing aid has been used.

```
  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1    PTR2   ROM
022273 36 36  0  0  1  0  0  1  0  0  01  114333 114303 0377713 0377732 000

      0   1   2   3   4   5   6   7     MEM 0:0
R00  000 012 265 230 273 044 150 204    026 000 011 210 303 030 011 210      C
R10  242 200 350 212 371 000 001 000    153 031 305 031 266 031 247 031   k E 6 '
R20  044 044 233 230 140 011 236 200    342 207 022 210 022 210 106 251   b     F)
R30  237 200 034 000 075 210 320 230    070 204 230 136 262 001 377 251   8   ^2 ■)
R40  110 233 230 001 000 000 044 044    340 040 262 030 377 321 000 140   ` 2 ■0 `
R50  000 051 000 000 000 000 000 000    365 012 262 065 210 261 014 140   v 25 1 `
R60  000 000 000 000 000 000 357 012    036 306 000 000 316 274 011 316    F  N< N
R70  016 316 000 000 000 000 000 000    030 030 230 316 306 207 117 220      NF 0
```

Output at a breakpoint includes:

1. The following CPU status indicators:

   PC: The setting of the program counter stored in registers R4 and R5. When execution is resumed, it will begin at the address specified by the PC.

   DR: Contents of the current data register pointer.

AR: Contents of the current address register pointer.

OV: Status of the overflow flag.

CY: Status of the carry flag.

NG: Status of the MSB (most significant bit), used to indicate a negative quantity.

LZ: Status of the LDZ (left significant zero) flag.

ZR: Status of the zero flag.

RZ: Status of the RDZ (right digit zero) flag.

OD: Status of the LSB (least significant bit), used to indicate an odd quantity.

DC: Setting of DCM (decimal) flag. Used to indicate decimal or BCD mode.

E: Contents of the E (extend) register. This will be a quantity between Ø and 17 octal.

BKP1: Indicates absolute address where breakpoint 1 is currently set.

BKP2: Indicates absolute address where breakpoint 2 is currently set.

PTR1: Indicates address of extended memory pointer 1.

PTR2: Indicates address of extended memory pointer 2.

ROM: Indicates number of ROM which was selected when the breakpoint occurred.

2. The contents of 1ØØ (octal) RAM or ROM locations are output beginning with the octal address specified in the last executed MEM and will continue for 1ØØ octal bytes. If no MEM was executed, 1ØØ (octal) bytes of memory will be output beginning with zero. The default ROM number is zero unless previously indicated. If MEM was executed, 1ØØ octal bytes will be output starting with the address of MEM.

3. Contents of CPU registers Ø through 77.

4. Memory contents in ASCII.

CLR breakpoint number

After CLR is displayed (as a result of typing "C"), the user can type 1 [END LINE] to clear BP1 or 2 [END LINE], to clear BP2. After CLR is displayed pressing [END LINE] or typing a number other than 1 or 2 will clear both breakpoints.

The CLR functions can be used any time execution has been halted, whether or not it has been halted by a breakpoint.

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

This command dumps the contents of computer RAM or ROM memory to the current CRT IS device beginning with the octal address selected. One-hundred octal bytes are dumped unless another parameter was input. The MEM function can be used after execution has been halted by a breakpoint.

The ROM number if included, is an octal value of selected plug-in ROMs from which memory is dumped. Default value for the ROM number is system ROM Ø, if no other ROM number has been selected.

The output shows the octal representation of the bytes in memory and the ASCII representation of the bytes.

If there are numeric entries after the "=" sign, memory is not dumped; the contents of memory locations beginning at the octal address specified are changed to the octal values after the "=" sign. The memory locations must be in RAM. The contents of one succeeding memory location are changed for each value specified after the "=" sign. The number of bytes, if included is disregarded in this case.

Examples: MEM 1Ø33ØØ
Dumps contents of 1ØØ octal bytes of memory to the CRT IS device, beginning with memory location 1Ø33ØØ.

MEM 1Ø33ØØ,2Ø
Dumps contents of 2Ø octal bytes of memory to the CRT IS device, beginning with memory location 1Ø33ØØ.

MEM 6Ø2ØØ: 4Ø,2ØØ
Dumps contents of 2ØØ bytes of the assembler ROM (ROM #4Ø) to the CRT IS device, beginning with memory location 6Ø2ØØ.

MEM 1Ø5ØØØ = Ø,Ø,Ø,15
Loads memory locations 1Ø5ØØØ, 1Ø5ØØ1, and 1Ø5ØØ2 with zeros, and loads location 1Ø5ØØ3 with 15 octal.

PC= octal address

Changes contents of program counter stored in CPU registers R4 and R5 to the specified address, and dumps CPU status and memory contents exactly as when a breakpoint (BKP) is executed. When execution is resumed, it will begin at the address now specified by the contents of the program counter (PC).

Example: PC = 3477 (Sets the PC to resume execution with byte 003477.)

REG number of CPU register = value

Changes contents of specified CPU register to the value given, and dumps CPU status and memory contents as when a breakpoint (BKP) is executed. Value may be octal, decimal, or BCD.

```
Example:  REG 35 = 31 (Changes contents of register R34 to 31 octal.)
          REG 36 = 19C (Changes contents of register R36 to BCD 19.)
          REG 37 = 25D (Changes contents of register R37 to 25 decimal.)
```

STEP

Although STEP is not a command, it is a typing aid which executes the next complete machine code instruction (not just the next byte). Beginning with the location currently addressed by the PC, it halts and dumps CPU status and memory contents like a breakpoint.

TRACE octal, decimal, or BCD value

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 status is output to the current CRT IS device. When execution halts, the CPU status and memory contents are output as at a breakpoint. Because of the internal coding of the system monitor, the address of BKP1 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).

To halt execution during TRACE, press any key. Repeatedly pressing a key may be necessary to halt TRACE.

Example: TRACE 1Ø output

```
  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
022274 46 36  0  0  1  0  0  1  0  0  01  022273 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
022275 46 36  0  0  0  0  0  0  0  0  01  022274 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
021636 46 36  0  0  0  0  0  0  0  0  01  022275 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
021637 46 36  0  0  0  0  0  0  0  0  01  021636 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
021640 46 36  0  0  0  0  0  0  0  0  01  021637 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
021641 20 36  0  0  0  0  0  0  0  0  01  021640 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
021642 20 10  0  0  0  0  0  0  0  0  01  021641 114303 0377713 0377732 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2   PTR1    PTR2    ROM
021643 20 10  0  0  0  0  0  1  0  0  01  114333 114303 0377713 0377732 000

       0   1   2   3   4   5   6   7     MEM 0:0
R00   000 012 265 230 243 043 155 204    026 000 011 210 303 030 011 210       C
R10   242 200 350 212 371 000 001 000    153 031 305 031 266 031 247 031   k E 6 ´
R20   040 044 233 230 140 011 236 200    342 207 022 210 022 210 106 251   b     F)
R30   237 200 034 000 075 210 316 230    070 204 230 136 262 001 377 251   8  ^2 ■)
R40   110 233 230 001 000 000 030 056    340 040 262 030 377 321 000 140   ` 2 ■0 `
R50   000 051 000 000 000 000 000 000    366 012 262 065 210 261 014 140   v 25 1 `
R60   000 000 000 000 000 000 357 012    036 306 000 000 316 274 011 316    F  N< N
R70   016 316 000 000 000 000 000 000    030 030 230 316 306 207 117 220      NF 0
```

PTR1= octal value

Changes pointer address.

PTR2= octal value

Changes pointer address.

WRITING BINARY PROGRAMS

6.1 Program Structure

An assembly language program is required to have a table of five pointers, or addresses, to tell the system where important parts of the program are. The system will use these pointers to find the table of keywords which the binary program implements and the associated routines to execute each of those keywords. This structure called the program shell is shown on the next page.

```
       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
```

The shell consists of the following parts:

1. The program control block.

2. Label definitions describing the locations of the tables that will allow the system to hook into the binary program. The following addresses must be included in this order:

   1. Run time routine table.
   2. ASCII keyword table.
   3. Parse routine table.
   4. Error message table.
   5. Initialization routine address.

3. The actual tables that have been defined previously. They must contain the addresses of the routines that will be performed.

   - The parsing routines will tell the system how to check a keyword for the proper syntax and parameters, and how to convert it to the internal RPN token format.

   - The actual translation of the keywords into machine operations is done by the run time routines whose addresses are defined in the run time table.

   - A marker, two bytes containing 377, must be set directly after the run time and parsing routine tables. When a binary program is loaded, this marker tells the system to assign an absolute address to all routines. All other addresses (routine references) are relative to the beginning of the program.

   - To let the system know which character strings will be the keywords, an ASCII table must be created to specify the keywords.

   - An error message table allows assembly language programs to specify custom error messages.

   - The code for a special initialization routine that is to be executed during initialization of the system, as at power-on, reset, allocation, and deallocation times. Refer to Initialization Hooks in section 3.

4. The routines that will actually do the operations required for defining and executing the new BASIC keyword must come after the tables.

The system will use the structure of the program shell to access the routines in the program. If a mistake is made in the structure, then the system cannot run the program.

The labels that are used to reference routines and routine tables can be any name as long as the names of routines in the tables correspond with the names of the routines themselves.

In addition, after the execution of a routine, control must be passed back to the system by executing a return. A return may be included after every routine.

Control Block

The program control block is 40 (octal) bytes long and is required to tell the system important things including:

- The first four characters in the name of the binary program.
- The length of the program in bytes, including the control block.
- The type of file is contained in the seventh byte. The format of the bits in this byte are as follows:

Bit	Meaning
0	000=BASIC Main Program
1	001=BASIC Subprogram
2	002=Binary Program
3	Undefined
4	Undefined
	Undefined
5	Undefined
6	0=Option base 1
	1=Option base 0
7	0=No COMMON
	1=COMMON

- The binary program number.
- The name of the file in mass storage (up to 10 characters).
- Six bytes required by the system.
- The base address of the first byte of the control block.

The control block is generated by the NAM instruction, which specifies the program name and number.

The program listed below is used in examples throughout this section.

```
1000          NAM 167,TEST
1010          DEF  RUNTIME
1020          DEF  KEYWORDS
1030          DEF  PARSING
1040          DEF  ERMSG
1050          DEF  INIT
1060 RUNTIME  BYT 0,0
1070          DEF  TEST.
1080 PARSING  BYT 0,0
1090          DEF  TESTPARS
1100 ERMSG    BYT 377,377
1110 KEYWORDS ASP "TEST"
1120          BYT 377
1130 INIT     RTN
1140 TESTPARS LDM R56,=0,371
1150          LDM R55,=PTR2-
1160          STMI  R55,=PTR2-
1170          JSB  =SCAN
1180          RTN
1190          BYT 241
1200 TEST.    JSB  =STBEEP
1210          RTN
1220 SCAN     DAD 21110
1230 PTR2-    DAD 177715
1240 STBEEP   DAD 10441
1250          FIN
```

Example: The program TEST is 107 (octal) bytes long and contains the following NAM statement and control block.

```
1000          NAM 167,TEST
```

<table>
<tr><td colspan="2">0
TEST</td><td>5
[LENGTH]</td><td>7
TYPE</td><td>8
BPGM #</td></tr>
<tr><td colspan="5">10
[NAME OF FILE AS ON DISC</td></tr>
<tr><td>20
DRIVE]</td><td>22
[BASE ADDRESS IF ABS]</td><td colspan="3">24
[UNDEFINED]</td></tr>
<tr><td>30
[LAST BYTE ADDRESS]</td><td>32
DEF RUNTIME</td><td>34
DEF ASCIIS</td><td colspan="2">36
DEF PARSE</td></tr>
</table>

Memory Contents	ASCII Representation
124 105 123 123 107 000 002 167	TESTG
124 105 123 124 102 040 040 040	TESTB
040 040 000 000 000 000 000 000	
064 221	

The first four bytes contain the ASCII representation for the name TEST. The next two bytes, with the least significant byte first, contain the length of the binary program in bytes. The type of file that the program is stored under is represented in eight bits (one byte), and the binary program number is stored in the last byte. The next 10 (decimal) bytes show the ASCII representation for the file name under which the program is stored, with ASCII blanks (040) to fill the rest of the bytes. The following six bytes are undefined, and the last two bytes contain the address of the first byte in the binary program.

System Table

The system uses this table to locate the routines and tables it will need to interpret the binary program. The system table must always be present in a binary program and must always define the subsequent tables in the proper order. During operations the system will need to have the address of a routine to handle parsing, initialization, execution, or error conditions. It will expect the address to be at the proper location as shown below:

Bytes From Base Address	Sample System Table
32	DEF RUNTIME
34	DEF KEYWORDS
36	DEF PARSING
40	DEF ERMSG
42	DEF INIT

When the system looks for a run time routine, it will add 32 (octal) to the base address of the program and access the run time routine table at run time. Likewise, it will add 34 (octal) to the base address to find the parse routine table, and so on. The system will expect the tables and the initialization routine to be in exactly these places in the program.

Placement of Binary Program Routine Tables

The addresses in the parse and run time routine table will be made absolute by the system when the program is executed. To indicate the end of the tables whose addresses will be absolute, the system looks for two bytes of 377's. Only the parsing routine and run time routine tables are required to have absolute addresses, so all other routine tables must follow the two bytes of 377's.

ASCII Keyword Table

The system will check a binary program for a BASIC keyword before it will try to process the keyword. In the ASCII table, all of the key words are arranged sequentially. When a BASIC statement is entered into the CRT, the system attempts to match the characters with a keyword in the table. The order of the keywords will affect the parsing and execution of the keyword, as the first keyword in the table will be processed by the first parsing routine in the parsing routine table and executed by the first run time routine in the run time routine table.

The system attempts to find a match by comparing each character in the table with each character in the keyword until it reaches a character with the most significant bit set. This indicates the end of a keyword, and, if no match has been found, the system assumes that the next character begins a new keyword and increments the number of the token. The search stops when a match has been found or a byte containing 377 is found.

Example: The following code creates an ASCII keyword table with one keyword, TEST. The ASP instruction creates an ASCII string with the most significant bit set on the last character, and the BYT 377 instruction signifies the end of the ASCII keyword table.

```
1110 KEYWORDS  ASP "TEST"
1120           BYT 377
```

Parsing Routine Table

If the system accesses a BASIC statement keyword that a binary program has listed in the ASCII keyword table, it will use the parsing routine provided in the program. Functions will be parsed by the system. The position of the keyword in the ASCII keyword table determines which parsing routine will need to be executed. If the keyword does not need to be parsed, then the corresponding position in the parsing routine table must be filled with two bytes of 0's to reserve the space corresponding to the ASCII keyword table.

The system will always skip the first two bytes after the location of the parsing routine table. The next two bytes are used as the address of the first parsing routine.

Example: The following parsing routine table has only one routine, TESTPARS, to parse the keyword TEST.

```
1080 PARSING   BYT 0,0          Two dummy bytes
1090           DEF TESTPARS     First parsing routine
```

Run Time Routine Table

Each keyword also has a run time routine associated with it. More than one run time routine may be listed in the table, so the system distinguishes between them in the same way as in the parsing routine table. When the system encounters a keyword that is listed in a binary program, it passes control to the proper routine corresponding to the position of the ASCII keyword in the keyword table.

Example: The run time routine table for the program TEST contains one routine address "TEST." which corresponds to the keyword "TEST" and the parsing routine TESTPARS.

```
1060 RUNTIME  BYT 0,0          Two dummy bytes
1070          DEF TEST.        First run time routine
```

Error Message Table

When an error is flagged in XCOM, the executive loop calls an error reporting routine that displays the error message. If the error number is less than 128 (200 octal), then it is a ROM error message and the bank-addressed ROM is selected whose number is in RAM location ERRROM. If the error message number is greater than 128 (200 octal), then the message will be from the binary program whose number is in ERRBP#.

The error message table is similar to the ASCII keyword table. It is constructed of entries which are strings of ASCII characters, the last character of each string having the most significant bit set. The table is terminated by a BYT 377.

Error messages in ROMs are numbered 0 through 177 (octal). Binary programs are numbered from 377 down to 200 (octal). The first nine error messages for ROMs and binary programs are for default errors. They will give only warning messages if defaults are on (refer to the owner's manual). The other error messages will always display the appropriate error message. Example error message table:

```
10 ERMSG     BYT 200,200,200,200        !!NINE DUMMY BYTES (377,367),
20           BYT 200,200,200,200,200   !! WITH THE MSB SET
30           ASP "SYNTAX-CHECK KEYWORD." !!ERROR 366 OCTAL
40           ASP "ROW OUT OF RANGE, >16." !!ERROR 365
50           ASP "COL OUT OF RANGE, >32." !!ERROR 364
60           BYT 377
70 INIT      BSZ 0
80           RTN
```

Initialization Routine

The program TEST has no need to initialize pointers, hooks, or flags. Therefore, the INIT routine returns control to the system. For times when further initialization is needed, refer to paragraph 3.5.

```
1130 INIT RTN
```

External Address Table

If any of the system locations have been used in the program, a table must be included to define the labels as absolute addresses.

Example: In the program TEST the addresses SCAN, PTR2-, and STBEEP are used and must be defined for the system.

```
1220 SCAN      DAD 21110
1230 PTR2-     DAD 177715
1240 STBEEP    DAD 10441
```

6.2 Attributes

Attributes define the type of a token. The system uses the attribute type to determine how parsing is to occur, how allocation and deallocation are to be performed, and how decompiling is to be done. The system is told how the keyword is to be handled at these times. The attributes must be defined immediately before the run time code in the program memory as shown in line 90:

```
1190           BYT 241
1200 TEST      JSB=STBEEP
1210           RTN
```

There are two types of attributes: primary and secondary. All keywords have primary attributes, but only functions have secondary attributes. The secondary attributes tell how many and the type of parameters the function will need and may occupy one or more bytes.

Attribute Location

The attributes must be placed directly before the run time routine code. The primary attributes must be the first byte before the run time routine. The secondary attributes would precede from the first byte to the last byte. The system checks the attributes from the bottom up, starting with the primary attribute and ending with the last parameter.

The following program listing:

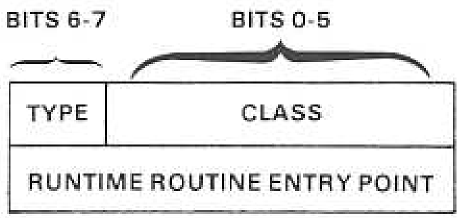
```
040 055        BYT 040,055
```

is the octal representation of these attributes:

```
055   Primary attribute - numeric function
040   Secondary attributes - two numeric parameters
```

Primary Attributes

The primary attribute consists of one byte of information containing the type of the keyword in the two most significant bits and the class of the keyword in the next six bits as shown:



The user must define the primary bits in order to tell the system exactly how he wants the system to recognize the keyword. For instance, if the keyword is to be a numeric function its attributes would be:

```
Function    Numeric Function
  Ø Ø       1  Ø  1  1  Ø  1

   Ø               55
```

Type

Bits 7 and 6 define how the keyword may be used. A keyword may be a BASIC statement or another command for calculator mode. A BASIC statement may be defined as legal after a THEN or illegal after a THEN. System commands are BASIC statements used only in calculator mode. Functions may be used in BASIC programs or in calculator mode.

The codes for each of the types are the following:

Bits 7,6	Octal	Type
1 1	3	BASIC statement, illegal after THEN
1 Ø	2	BASIC statement, legal after THEN
Ø 1	1	System commands (nonprogrammable)
Ø Ø	Ø	Functions and others

Class

The class will give the system further information on how to process the keyword. The class should follow directly after the type of keyword. For example, a function that returns a number will be in the numeric function class. Keywords that are to be invisible when the program is decompiled have their own class. All BASIC statements that are not functions and all system commands are reserved words.

Example: The keyword "INPUT" uses two tokens to compile but only one shows when it decompiles. The first token puts the system into a pseudo-calculator mode to allow characters to be entered from the keyboard and outputs a "?." The other, which is hidden during decompiling, takes the system out of the pseudo-calculator mode and does the actual storing of the input values. The class of the second keyword keeps it from being printed in the program listing. The keyword "LOAD" is in the class of reserved words. Refer to the sample program LINPUT in section 7.

Useful Classes

Bits 5-Ø	Octal	Class
1 Ø Ø Ø Ø 1	41	Reserved words
1 Ø Ø 1 Ø Ø	44	Invisible at decompile time
1 Ø 1 1 Ø 1	55	Numeric function (such as, SIN, IP)
1 Ø 1 1 1 Ø	56	String function (such as, CHR$, VAL$)

Secondary Attributes

At parse time, if the system parser finds a match for a keyword in the binary program ASCII table, it will then check the attribute type. If the keyword is a statement, control passes to the binary program parse routine. If the keyword is a function, the secondary attributes determine the type and number of parameters to use.

One byte is needed if the function uses one or two parameters, and a second byte is needed if there are more than two but less than seven parameters. More parameters require more bytes. The first four bits of the first byte indicate the number of parameters that the function will accept. The next two bits define the type of the first parameter, and the second parameter is defined by the last two bits. Thereafter the consecutive pairs of bits define extra parameters.

Parameter Types

Type	Description
Ø Ø	Numeric
Ø 1	Numeric array
1 Ø	String
1 1	Strange

ØØ1Ø	ØØ	ØØ
2 Parameters	First	Second

6.3 Assembler Instructions

The instruction set is used to communicate between the assembly language programmer and the CPU. Assembly language instructions can move data, perform arithmetic operations, and execute other functions. There are two types of instructions: those which operate directly on the CPU and are translated into machine language; and pseudo-instructions which act as messages to the Assembler ROM.

The typical instruction is broken up into five fields. The first field is the line number, for the convenience of the programmer. When assembled, the program will not have line numbers, instead it will show the value of the instruction counter. The instruction counter is the offset in bytes from the start of the program. The next field is an optional label field. System labels may be defined in the global file. Other labels must be defined in the routine. The opcode comes after the label field and is the heart of the instruction because it tells the CPU or Assembler ROM what is to be done. Following the opcode is the operand(s) for the instruction, and at the end of the instruction or in the label field the programmer may place a "!" followed by a comment.

In assembler mode the system will automatically space the elements typed in the proper fields. The programmer has only to distinguish the fields by at least one space. The registers may be referred to by their octal numbers, and the system will add the "R" in its proper place.

Example: Line number 12Ø may be typed in as follows:

```
120  LDMD 46,22 !A MULTI-BYTE LOAD
```

After pressing [END LINE], it will appear in the program listing as:

```
120          LDMD R46,R22             !A MULTI-BYTE LOAD
```

Line Numbering

Each line of a program source code must begin with a line number (which will not appear in the assembled code). A line number may be up to 99999 and may be entered individually or automatically, using [AUTO] for automatic line numbering. When a program is assembled the line numbers will appear as relative addresses of the instructions, that is, the instruction location counter.

Labels

A label may be from one to eight characters long. The label field starts in the second space after the line number. A digit may not be used as a first character, and no spaces may be used in a label because a space denotes the end of a label. When variable storage is needed in the program, a label may be used after the run time routine. To simulate control loops and branch execution, a label may be used to designate the location of the jump.

Opcodes

The 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. Blanks are not allowed within the opcode field.

Opcodes may be single-byte, multi-byte, or pseudo-operations. The pseudo-operations may act upon bytes but are only messages to the Assembler ROM and do not generate executable code.

Operands and Addressing

Depending upon the kind of instruction to be performed, the operand may be a register, a label or address, a pointer to a value, or a relative location which must be offset by an absolute address. The DRP will point to the register that will be operated upon according to the opcode. If the opcode calls for direct addressing, where the value is at a location outside of the CPU register bank, the operand will contain the address of the value in memory. If the opcode calls for direct addressing, where the value is pointed to by a label that is located outside of the CPU register bank, the operand will contain the address of the pointer in memory.

Indexed addressing can be used to access an area of memory by adding a base address to an offset such as in table searching. The absolute address of a label can be obtained by adding BINTAB to the relative address of the label.

Comments

A comment must begin with an exclamation point "!." A comment may be typed beginning in the first or second space after the line number or one or more spaces after the other elements of the instruction. Comments may be as long as needed, though the limit is 160 characters per line.

Constants

Constants may 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 system assumes octal values. Constants will be stored as one or more bytes, depending on whether it indicates a single- or multi-byte operation. After the program is assembled constant values are placed immediately after the machine code.

Syntax and Explanation

Each of the opcodes are discussed in detail in the next three subsections. The opcode is shown above its explanation, then following the explanation is an example of how the instruction may be used.

The first two letters of the opcode signify its operation, but the designation for a single-byte, "B," or multi-byte, "M," operation must be added at the end. In addition, if a type of addressing other than register immediate is needed, then the letter for that addressing mode must be added, "D" for direct or "I" for indirect. Instructions using direct or indirect addressing will have opcodes of four characters. A register being used for indexing must be entered in the operand field with an "X" instead of an "R." Pseudo-instructions always have opcodes of three characters.

The examples are designed to give the programmer a few hints for using the instructions in binary programs and clarify some points about the syntax of the instruction set.

Syntax Guidelines

LDB Instructions shown in capital letters must be entered exactly as shown (in either upper- or lower-case).

[ ]	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.
( )	Contents of.
—	Complement.
B/M	Single- or multi-byte instruction.
AR	Address register location. Location of first byte addressed by the ARP. Can be a register, R*, or R#.
DR	Data register location. Location of first byte addressed by the DRP. Can be a register, 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) which must be 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.

LOAD/STORE Instructions

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

LD
CPU Instruction

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

```
Format:  LDBA DR, operand      single-byte
         LDMA DR, operand      multi-byte
```

ST
CPU Instruction

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

```
Format:  STBA DR, operand      single-byte
         STMA DR, operand      multi-byte
```

Addressing Modes

The CPU allows 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) instruction have access to all addressing modes except stack addressing, and they are used here for illustration. For a list of the addressing modes used by a particular instruction, refer to appendix B.

Most addresses are referred to as two-byte quantities. Because addresses are two consecutive bytes, only the least significant byte is referenced. For instance, the address register (AR) is actually a single byte within the CPU register bank that is pointed to by the address register pointer (ARP). When the address register contains an address, the CPU register pointed to contains the least significant byte of the address. The next register (ARP + 1) contains the most significant byte of the address.

The multi-byte feature of the CPU allows data to be manipulated in quantities of one to eight bytes. Therefore, in the following descriptions, only the address of the first byte is specified.

In the following descriptions, the effective address (EA) points to the first byte of data to be affected by the instruction.

Register Mode

This mode allows the CPU registers to contain addresses as well as data. There are three types of register addressing: register immediate, register direct, and register indirect.

Register Immediate

Examples:

LDB R36,R32	Loads contents of R32 into R36.
STM R40,R50	Stores the contents of R40-R47 into R50-R57.

Register Direct

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 the location addressed by R50-R51.

Register Indirect

Examples:

LDBI R36,R32	If R32 contains 105731, and location 105731 contains 110437, the contents of 110437 is loaded at location R36.
STBI R36,R32	If R32 contains 105371, and 105731 contains 110437, then the contents of R36 is stored at location 110437.

Literal Mode

The operand is a literal quantity stored in memory immediately following the opcode. A literal string, ten octal bytes or less, is a BCD, octal, or decimal constant or a 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 a mismatch. Literal mode includes literal immediate, literal direct, and literal indirect forms of addressing.

Literal Immediate

Examples:

LDB R36,=10D	Loads 10 decimal (12 octal) into CPU register R36.
LDM R40, =0,0,0,0,0,0,0,120	Loads 120 octal (a floating point 5 in BCD format) into register R40-R47.
LDM R32,=LABEL	Loads R32-R33 with the relative address of LABEL.

Literal Direct

Examples:

LDBD R34,=ROMFL	Loads the contents of the memory location addressed by the label ROMFL into CPU register R34.
STMD R74,=CHIDLE	Stores the contents of CPU register R74 through R77 into four memory locations beginning with the location addressed by the label CHIDLE.

Literal Indirect

Example:

STBI R3Ø,=ADDR	Stores the contents of CPU register R3Ø 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

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 R2 and R3. After an index instruction, these registers contain the effective address (the sum of the base and the offset). Neither the original base nor the offset is altered in memory. There are two types of indexed addressing: index direct and index indirect.

Index Direct

Example:

LDBD R36,X3Ø,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 R3Ø-R31 and the address TABLE.

Index Indirect

Example:

STMI R36,X3Ø,OFFST	Stores the contents of CPU register R36 and R37 in memory, beginning with the location addressed by another memory location which is addressed by CPU registers R2 and R3. Registers R2 and R3 contain the sum of the address in R3Ø-R31 plus the offset specified by the label OFFST.
STBI R36,X34,66	Stores the contents of R36 in the location addressed by R2 and R3 (sum of the address in R3Ø-R31 plus 66).

Stack Instructions

In stack addressing, a register pair serves as a pointer to the stack in memory. A load or store is performed at the top of the stack, and the register pair is decremented or incremented to the new top of the stack. Instructions push and pop are available to push data onto and pop data from stacks in the main memory. These stacks can be addressed using direct or indirect addressing.

PU

Pushes single byte or multi-byte using direct or indirect addressing. The stack pointer is incremented (increasing stack) or decremented (decreasing stack).

Examples:

PUBD R32,+R12	Pushes single byte from R32 onto the R12 stack. The stack pointer is incremented.
PUBI R32,-R46	The stack pointer is first decremented and then the single byte contained in R32 is pushed onto the R46 stack.

PO

Pops single byte or multi-byte off stack using direct or indirect addressing. The stack pointer is incremented (increasing stack) or decremented (decreasing stack).

POBD R32,+R2Ø	Pops single byte contained in R12 onto the R2Ø stack. The stack pointer is incremented after the operation.
POBD R32,-R2Ø	The stack pointer is first decremented and then R32 is loaded with the byte pointed to by R2Ø.

Stack Addressing

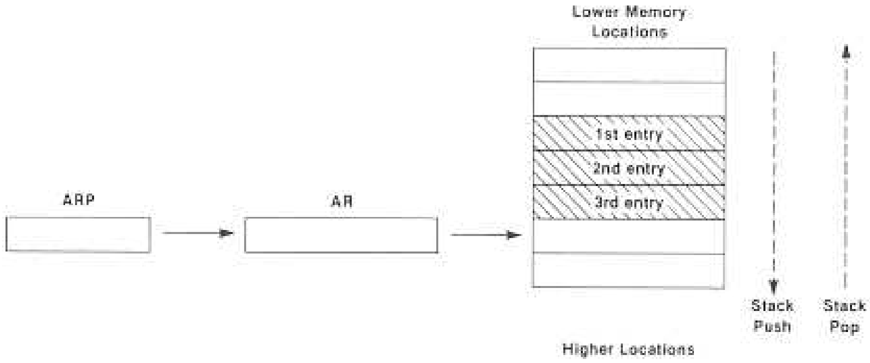
You can address a stack from nearly any CPU register pair. Registers R6 and R7 are hardware-dedicated and always point to the subroutine return stack, a fixed stack of 512 bytes. A subroutine jump will automatically push an address onto this stack and a return will load the program counter with the address on the top of the stack, causing execution to begin at that address on the next cycle. The R6 stack is also affected by SAD and PAD instructions (save and restore status), which push three bytes onto the R6 stack and remove them respectively.

Another stack used by many of the system routines at run time is the R12-R13 operation stack. This stack is used to pass parameters between system routines. The documentation for each routine using this stack describes what the routine expects on the R12 stack and what it leaves after it has finished.

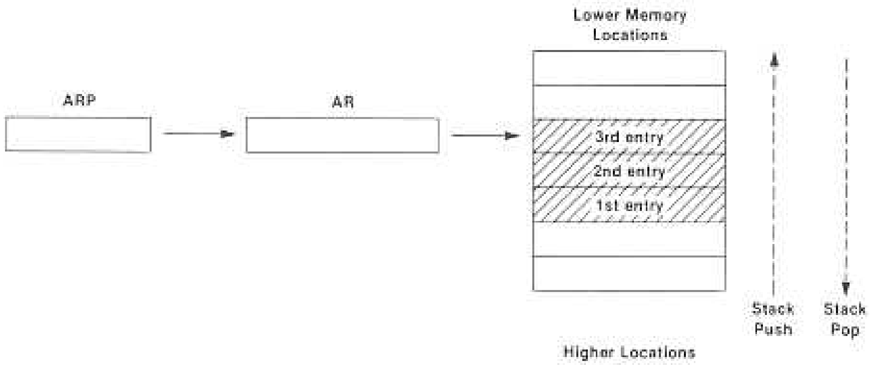
Stacks may be increasing or decreasing. An increasing stack is one which is filled in the direction of higher memory locations and from which data is removed in the direction of lower memory locations. In a decreasing stack, data is pushed in the direction of lower memory locations, and taken off in 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 the ARP equal to the location of that stack pointer.

For an increasing stack, the AR must point to the next location available 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.

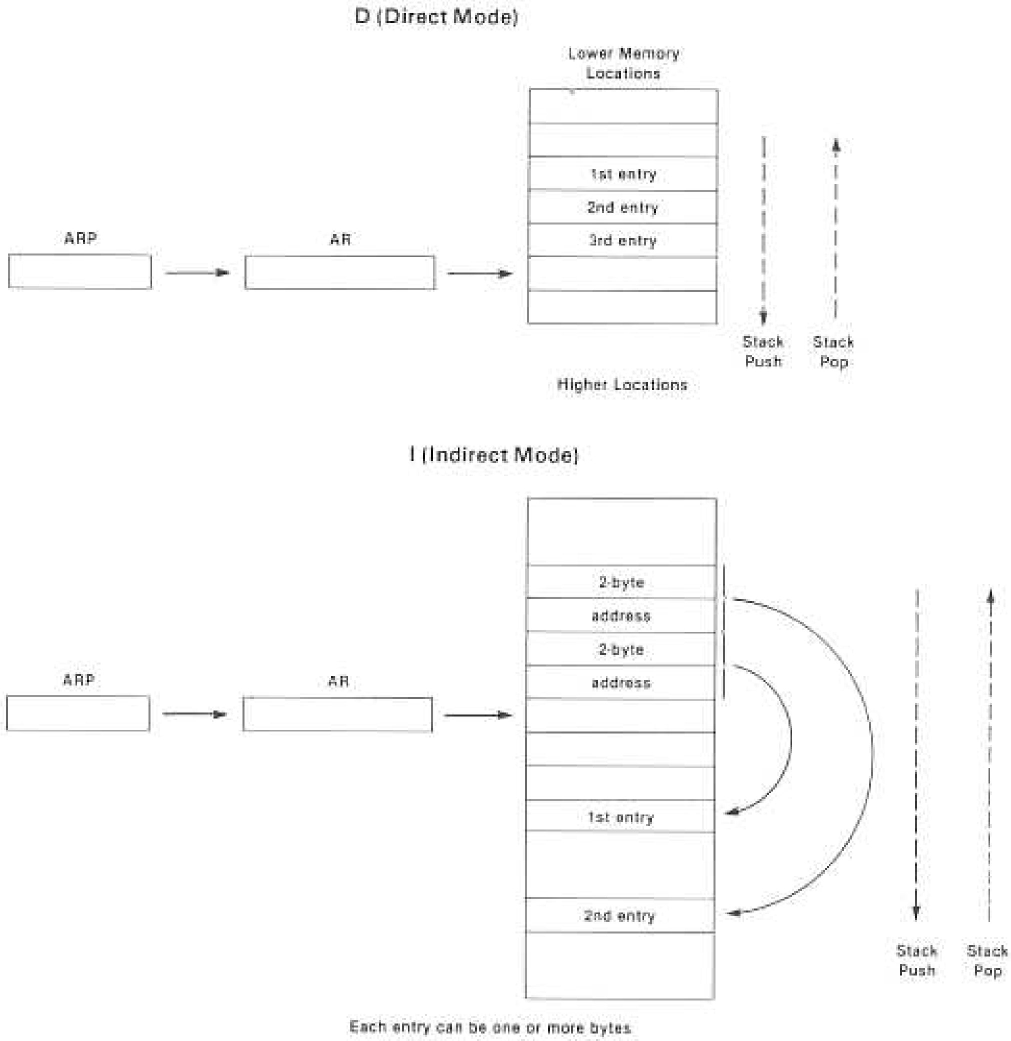
For a pop from a decreasing stack, the AR points to the location of the data to be removed. Following the removal, the AR is incremented by the number of bytes moved. For a push operation onto a decreasing stack, the AR is first decremented by the number of bytes to be stored on the stack. Then the data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

Stack Indirect

In this 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. The effective address is then loaded into the CPU register designated by the DRP.

```
PUBD DR,+AR     Push byte direct with increment.
```

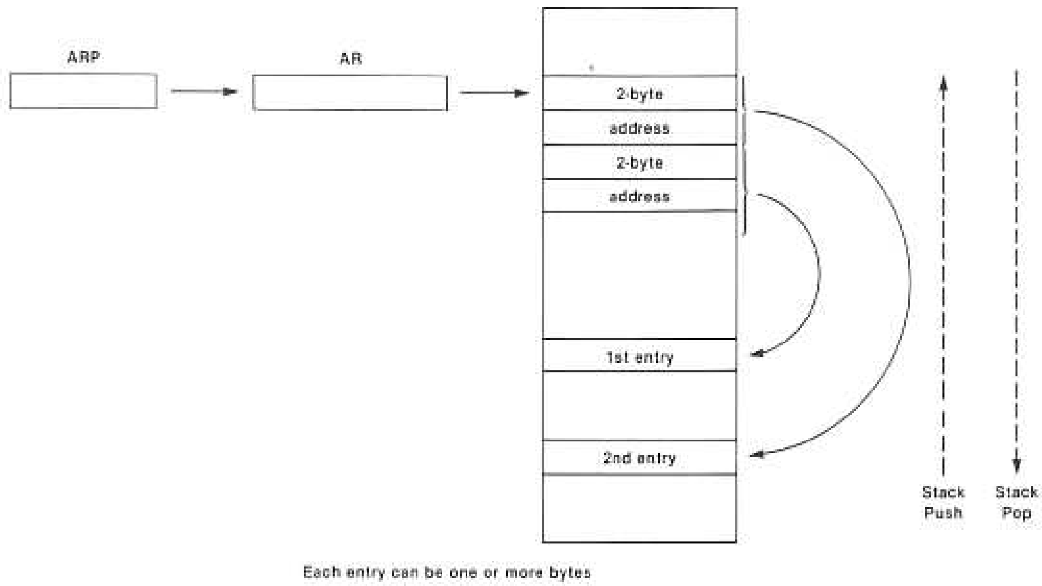


INCREASING STACK

The instructions available for use with an increasing stack are:

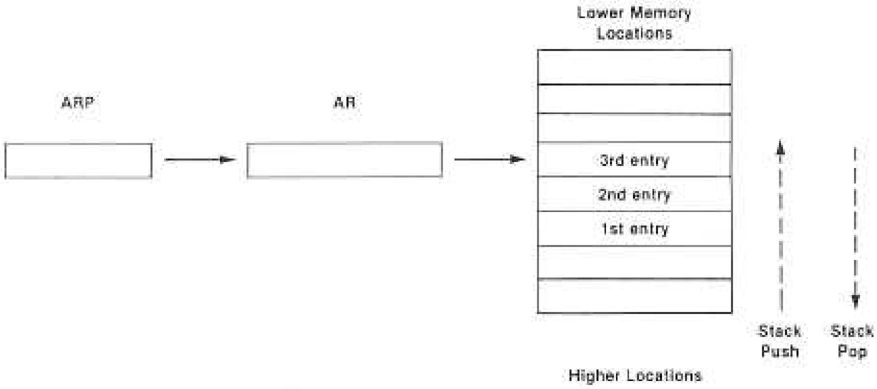
PUBD DR,+AR	Push byte direct with increment.
PUMD DR,+AR	Push multi-byte direct with increment.
PUBI DR,+AR	Push byte indirect with increment.
PUMI DR,+AR	Push multi-byte indirect with increment.
POBD DR,-AR	Pop byte direct with decrement.
POMD DR,-AR	Pop multi-byte direct with increment.
POBI DR,-AR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

I (Indirect Mode)



DECREASING STACK

D (Direct Mode)



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 decrement.
POBI DR,-AR     Pop byte indirect with decrement.
POMI DR,-AR     Pop multi-byte indirect with decrement.
```

Arithmetic and Logical Instructions

AD

Add may be used to combine the value of the data register and the contents of the operand. This operation may be performed on single bytes or multiple bytes, and direct addressing or constants may be used. In BCD mode addition will take place using four-bit digits. The result is always stored in the data register.

Example:

ADB R2Ø,R3Ø	Adds the contents of R3Ø to R2Ø and places the result in R2Ø.
ADMD R2Ø,=BINTAB	Takes the location of the beginning of the binary program and adds it to the value in R2Ø, R21. The result is stored in R2Ø, R21.

AN

Each bit in the data register is compared to the corresponding bit in the operand. If the bits being compared are both 1, then the result is a 1. If either bit is Ø, then the result is Ø. The operand may be a value in memory that is addressed directly. Although this instruction is available only for multi-byte operations, single-byte operations are possible with the DRP set to a boundary register.

Example:

ANM R2Ø,R3Ø	Converts all of the 1's in R2Ø-R21 to Ø's if the same bits in R3Ø-R31 are Ø's.

If R2Ø-R21 contain:	1 1 Ø 1 1 Ø 1 1	Ø Ø 1 Ø 1 1 Ø Ø
and R3Ø-R31 contain:	Ø 1 Ø Ø Ø 1 1 1	Ø Ø Ø Ø 1 Ø 1 1
then the result is:	Ø 1 Ø Ø Ø Ø 1 1	Ø Ø Ø Ø 1 Ø Ø Ø

CM

The compare is used to simulate the logical operations of a high level language. It is done by subtracting the operand from the data register and setting the appropriate status indicators; the result of the operation is not stored. In binary mode the subtraction is two's complement, and in BCD mode the subtraction is ten's complement. Compares may be either single- or multi-byte operations, and direct addressing may be used. When used previous to a logical jump, an IF-THEN BASIC statement may be simulated.

In order to simulate the relation:

DR<AR	CMM DR,AR JNC LABEL	CY flag should be Ø Jump if DR<AR
DR>=AR	CMM DR,AR JCY LABEL	CY flag should be 1 Jump if DR>=AR
DR=AR	CMM DR,AR JZR LABEL	ZR flag should be 1 Jump if equal
DR#AR	CMM DR,AR JNZ LABEL	ZR flag should be Ø Jump if not equal

The jump instructions JNZ, JZR, JCY, and JNC are explained later in this section.

OR

Each bit in the data register is compared to the corresponding bit in the operand. If either bit is a 1, then that bit in the data register is set to 1. Otherwise the bit in the data register is set to Ø. This logical operation may be performed on single bytes or multiple bytes, but must use register immediate addressing only.

Example:

ORB R2Ø,R3Ø	Leaves a 1 in R2Ø if the corresponding bit in R3Ø is set.

If R2Ø contains:	Ø Ø 1 Ø 1 1 Ø Ø
and R3Ø contains:	Ø Ø Ø Ø 1 Ø 1 1
then the result is:	Ø Ø 1 Ø 1 1 1 1

SB

Subtraction is simulated by adding the complement of the operand to the data register. Ten's complement is used in BCD mode, and in binary mode two's complement is used. The result of the subtraction is stored in the DR. The operand may be addressed immediately or directly, and can be a single- or multi-byte instruction. The CY flag is set to 1 if the result is positive and cleared if the result is negative.

Example:

SBM R2Ø,R3Ø	In binary mode, takes the two's complement of R3Ø-R31 and adds that to R2Ø-R21. The result is put in R2Ø-R21.

If R2Ø-R21 contain:	1 1 Ø 1 1 Ø 1 1	Ø Ø 1 Ø 1 1 Ø Ø
and R3Ø-R31 contain:	Ø 1 Ø Ø Ø 1 1 1	Ø Ø Ø Ø 1 Ø 1 1

then the complement of R3Ø-R31

1 Ø 1 1 1 Ø Ø Ø	1 1 1 1 Ø 1 Ø 1

is added to R2Ø-R21.

The result is:	1 1 Ø 1 1 Ø 1 1	Ø Ø 1 Ø 1 1 Ø Ø

The operation is done in binary mode. Since registers are shown in octal, the previous example would look like this:

	R2Ø	R21	R3Ø	R31
Before:	Ø54	333	Ø13	1Ø7

Two's complement

	R2Ø	R21	R3Ø	R31
Result:	223	Ø41	Ø13	1Ø7

Example:

SBB R2Ø,R3Ø	In BCD more, takes the ten's complement of the two digits in R3Ø and adds that to the two digits in R2Ø.

If R2Ø contains:	Ø Ø 1 Ø 1 1 Ø Ø
which in BCD are the decimal digits:	28
and R3Ø contains:	Ø 1 Ø Ø Ø 1 1 1
which in BCD is:	47
Then the ten's complement of R3Ø:	53
is added to R2Ø:	28
and the result in R2Ø is:	8Ø

XR

In the "exclusive or" logical operation the bit that corresponds in the data register is set to 1 when the bits being compared are not the same. When both bits are 1 or both bits are Ø, the bit in the data register is set to Ø. The CY and OVF flags are cleared.

Example:

XRM R2Ø,R3Ø — Compares the individual bits in R2Ø-R21 and R3Ø-R31. If they are not the same, sets that bit to 1 in the DR; otherwise it is set to Ø.

If R2Ø-R21 contain:	1 1 Ø 1 1 Ø 1 1	Ø Ø 1 Ø Ø 1 Ø Ø
and R3Ø-R31 contain:	Ø 1 Ø Ø Ø 1 1 1	Ø Ø Ø Ø 1 Ø 1 1

The result in R2Ø-R21 is:

1 Ø Ø 1 1 1 Ø Ø	Ø Ø 1 Ø Ø 1 1 1

Shift Instructions

All shift instructions can be done in BCD or binary mode. In BCD mode the shift will affect a BCD digit, or four bits, and in binary mode it will affect a binary digit, a bit. Shifts may also be single- or multi-byte operations, and the result of a shift will be determined by the nearest boundary in the direction of the shift. In single-byte shifts the boundary is actually the register being shifted, whereas in multi-byte operations the boundaries are those in the CPU register bank. In arithmetic and logical operations the boundaries are normally toward the higher-numbered registers. With shifts, the boundary may be to the left, higher-numbered registers, or the right, lower-numbered registers depending on whether you are shifting right or left.

Shifts are made into one of the shift registers: the E register or the CY flag. In BCD mode shifts are made into and out of the E register, and in binary mode shifts are made into and out of the CY flag.

EL

The extended left shift will take the most signficant digit, put it into the shift register, move the rest of the contents one digit to the left and put the previous contents of the shift register into the least significant digit.

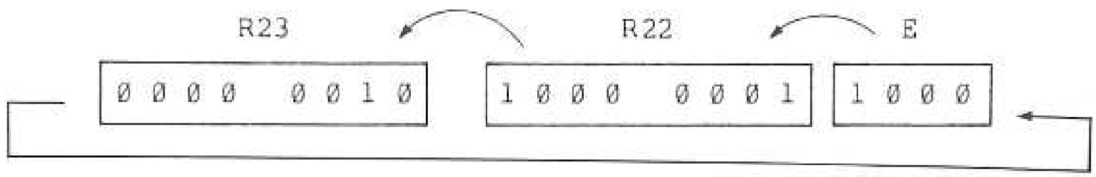
Example:

ELM R20	In BCD mode, shifts the most significant digit of R20-R21 (1000) into the E register. The other 12 bits will move left four bits, and the least significant digit will be filled with the previous contents of the E register (0001).

	R23	R22
If R20-R21 contain:	1 0 0 0 0 0 0 0	0 0 1 0 1 0 0 0
	80	2 8

and "E" contained previously: 0 0 0 1

then the shift would take place as follows:



ER

The extended right shift moves the least significant digit to the shift register and the contents of the shift register into the most significant digit. It works in the same way as the extended left shift except that the movement is toward the right boundary.

Example:

ERB R21 — In binary mode, shifts the least significant bit (LSB) to the CY flag, then moves the previous contents of the CY flag to the MSB position.

If R21 contains: [1 1 0 1 1 0 1 1]

and the CY flag is: [1]

then the result would be: CY [1] [1 1 1 0 1 1 0 1]

LR

When a logical right shift is performed, the LSB is moved into the shift register and the MSB is cleared. The digit is maintained in the shift register and may be shifted back using the extended shift instructions.

Example:

LRM R21 — In binary mode, shifts the LSB into the CY flag and clears the MSB.

	R21	R20	CY
If R20-R21 contain:	1 1 0 1 1 0 1 1	0 0 1 0 1 1 0 0	
The result is:	0 1 1 0 1 1 0 1	1 0 0 1 0 1 1 0	0

LL

The logical left shift moves the most significant digit of the data register into the shift register and clears the least significant bit. If the shift causes a sign change then the OVF is set to 1.

Example:

LLM R3Ø	In binary mode, shifts the MSB of R3Ø into the CY flag and clears the LSB of R31.

		R31	R3Ø
If R3Ø-R31 contain:		Ø 1 Ø Ø Ø 1 1 1	Ø Ø Ø Ø 1 Ø 1 1

	CY	R31	R3Ø
then the result would be:	Ø	1 Ø Ø Ø 1 1 1 Ø	Ø Ø Ø 1 Ø 1 1 Ø

DC

The decrement is simulated by adding the complement of 1 binary in binary mode, to the quantity in the data register. The quantity may be single or multiple bytes.

Example:

DCB R31	Subtracts one from the quantity in R31.

	R31
If R31 contains:	1 Ø Ø Ø Ø Ø Ø Ø

	R31
then the result is:	Ø 1 1 1 1 1 1 1

The OVF flag is set to 1, because the sign changed.

IC

When an increment is performed, 1 is added to the quantity in the data register. In BCD mode, the quantity is incremented by decimal 1, and in binary mode, it is incremented by a binary 1. In BCD mode the OVF flag is cleared (single- or multi-byte).

Example:

ICM R2Ø	In BCD mode, the decimal quantity in R2Ø, R21 is increased by 1.

	R21	R2Ø
If R2Ø-R21 contain:	1 Ø Ø 1 1 Ø Ø 1	Ø Ø 1 Ø Ø 1 Ø 1
which in BCD is:	9 9	2 5

	R21	R2Ø
then the result is:	1 Ø Ø 1 1 Ø Ø 1	Ø Ø 1 Ø Ø 1 1 Ø
which in BCD is:	9 9	2 6

NC

This complement instruction will give the nine's complement in BCD mode and the one's complement in binary mode. The nine's and one's complement are performed by taking the number of digits to be complemented and subtracting each digit individually from 9 in BCD mode and 1 in binary mode. The result is placed in the data register.

Example:

NCB R2Ø	In binary mode, flips all bits (one's complement operation).

	R2Ø
If R2Ø contains:	Ø 1 Ø Ø Ø 1 1 1
then the result is:	1 Ø 1 1 1 Ø Ø Ø

Example:

NCM R2Ø	In BCD mode, takes the nine's complement of the contents in R2Ø-R21 by subtracting each digit from a BCD 9 (1ØØ1).

	R21		R2Ø	
If R2Ø-R21 contains:	1 Ø Ø Ø	Ø Ø Ø Ø	Ø Ø 1 Ø	1 Ø Ø Ø
which in BCD is:	8	Ø	2	8

	R21		R2Ø	
then the result would be:	Ø Ø Ø 1	1 Ø Ø 1	Ø 1 1 1	Ø Ø Ø 1
which in BCD is:	1	9	7	1

TC

The contents of the data register is replaced by the two's complement in binary mode or the ten's complement in BCD mode. Two's and ten's complement is found by incrementing the one's or nine's complement. In BCD mode, the OVF flag is cleared.

Example:

TCM R2Ø — In binary mode, takes the two's complement of R2Ø-R21.

	R21	R2Ø
If R2Ø-R21 contain:	1 1 Ø 1 1 Ø 1 1	Ø Ø 1 Ø 1 1 Ø Ø
then the result would be:	Ø Ø 1 Ø Ø 1 Ø Ø	1 1 Ø 1 Ø 1 Ø Ø

TS

The status of the contents of the data register are tested, and the appropriate status indicators are set. The OVF and CY flags are cleared in all cases, and the E register is not affected. This instruction is a single- or multi-byte instruction. The status indicators are discussed in section 2.

Example:

TSM R2Ø	Will set the status indicators and clear the OVF and CY flags.

	R21	R2Ø
If R2Ø-R21 contain:	Ø 1 Ø Ø Ø Ø 1 1	Ø Ø Ø Ø 1 Ø 1 1
	LDZ	RDZ

the resulting flags will be set:

DCM	May be 1 (BCD) or Ø (binary).
E	Not affected.
CY	Cleared.
OVF	Cleared.
OD	Set to 1.
NG	Set to Ø.
Z	Set to Ø (since quantity is nonzero).
LDZ	Set to Ø.
RDZ	Set to Ø.

CL

The clear instruction permits the clearing of any byte or of any multi-byte portion of the CPU register bank. The DR is set equal to Ø and the flags CY and OVF are cleared.

Example:

CLB R47	Clears R47

JSB

When a subroutine jump is made, the control of the program is given to a set of instructions with the intention of returning to the program at the next instruction after the jump was made. In order to return, the program counter for the next instruction must be stored. This return location is pushed onto the R6-R7 stack, and when the RTN instruction is executed, it is loaded back into the program counter. A subroutine jump that is made to a relocatable address in a binary program must be indexed from the absolute start of the program (BINTAB).

Examples:

JSB =NUMVA+	Increments the program counter (PC) to the address of the next instruction after the JSB. That address is pushed onto the R6-R7 stack, and the PC is loaded with the address the jump is to be made to NUMVA+ (located at 22403). When the system executes a RTN, it pops the address of the next instruction off of the R6-R7 stack and loads that value into the PC.
JSB X14,ROUTINE	Makes a jump to ROUTINE by adding the value of ROUTINE as a label to the location of the start of the program (BINTAB) which is stored in R14-R15. In all other aspects it is the same as JSB=.

Jump Instructions

This group of instructions gives the capability for branching control to addresses that are defined by the label that the jump is being made to. If a condition is true, then the jump is made; otherwise, the jump is ignored and the next instruction is executed. These branching instructions use relative addressing. Labels that are used must be contained inside the program. The program counter (PC) is loaded with the value of the address, and program control moves to that location in the program memory. The maximum number of bytes that may be jumped is 177 octal (forward) higher-addressed bytes or 200 octal (backward) lower-addressed bytes.

Each conditional jump has a complement, except the jump on no overflow, which jumps on the opposite of the relation. For instance, the jump on negative is simply the opposite of the jump on positive and may be used in the same circumstances depending on the personal preference of the programmer. All of the jumps will be discussed.

JMP

The unconditional jump always occurs when executed. It is the only jump that does not check the status of any system flags.

Example:

JMP ALWAYS	Will always jump to ALWAYS, a location in the program.

JNO

Since the system has no jump on overflow, a jump on no overflow must be used for both cases. If the OVF flag is set to 1, then the jump is ignored and the next instruction will be executed. In the case of an overflow, the code after the jump instruction will perform the necessary steps, and then if necessary, continue the program.

Example: If a flag (E) is to be incremented when an overflow occurs:

	ADM R2Ø,R3Ø	Executes the operation that may set an overflow (OVF).
	JNO RESUME	If there has been no overflow, the program will begin at RESUME.
	ICE	If JNO is ignored, then an overflow has occurred, and the program increments the E flag.
RESUME	BIN	Resumes the program.

JPS, JNG

Jump on positive and jump on negative are made by checking the status of the most significant bit (NG) flag and taking the "exclusive or" of NG and the OVF. In the case of two positive numbers added together resulting in a negative number (NG=1), the jump on positive takes that into consideration and would jump because NG=1 and OVF=Ø and the "exclusive or" would be 1 and the jump would be made.

Example: If R2Ø contains Ø73 and R3Ø contains 125 then the addition:

ADDITION	ADB R2Ø,R3Ø	Adds Ø73 to 125 (octal) and sets NG=1 and OVF=Ø.
	JNG ADDITION	Since the exclusive OR of NG=1 and OVF=Ø is 1 and JNG expects it to be Ø, then the jump will not be made even though the NG flag says it is negative.

JOD, JEV

The least significant bit flag (OD) shows whether a number is even or odd. If the number is even, OD is set to 0 and JEV, jump on even, will take place. If the number is odd, OD=1, then JOD, jump on odd, will take place. This conditional jump works in binary and BCD modes.

Example: To find out if the 16-bit binary number stored in R36-R37 is a prime number, all even numbers may be ignored by the following code:

```
TSM R36          Checks to see if the number is even.
JEV NOTPRIME     Since the number is odd, it might be prime.
```

JZR, JNZ

When making comparisons and when decrementing a counter, the jump on 0 and jump on not 0 are useful. If two quantities are equal, comparing them will cause the ZR flag to be set to 1. To simulate a conditional IF-THEN statement, a comparison is made prior to the jump. To simulate a controlled FOR-NEXT loop, the loop counter is decremented and the conditional jump made.

Example: To simulate IF X=80 THEN RESUME (R20 contains 120 octal which is 80 decimal):

```
CMB R20,=120          Compares R20 to 120. Since they are
                      equal, the ZR flag is set to 1.

JZR RESUME            Since ZR=1, the jump is made to the
                      location RESUME.
```

To simulate the FOR-NEXT loop, the number of times that the loop will be executed is decremented and a check is made to see if that number of loops has been done.

Example: If R20 contains the number of times the loop is to be executed, then FOR X=1 to 20 would be:

```
DCB R20               After the statements have been executed,
                      R20 is decremented. If R20 is equal to
                      zero, the ZR flag is set to 1.

JNZ LOOP              If the loop has not been done the
                      specified number of times, it must be
                      done again starting at the beginning of
                      the statements (LOOP).
```

JCY, JNC

When the carry flag (CY) is set to 1, it indicates an addition has become too large for the register to handle. This happens often in subtraction and in comparisons. To simulate the statement IF-THEN with a "greater than or equal to" or "less than" relation, a compare is made between the values, and the CY flag is checked.

Example: If R2Ø and R3Ø contain the first and second numbers to be included in the compare, then the statement IF QUANTITY1 > QUANTITY2 THEN RESUME could be:

CMB R2Ø,R3Ø	Compares R2Ø to R3Ø by adding the negative of R3Ø to R2Ø and sets the status flags. If R2Ø is greater than or equal to R3Ø then CY=1. If R2Ø is less than R3Ø, CY=Ø.
JCY RESUME	Jumps to the location RESUME if R2Ø is greater than R3Ø (CY=1).

JEZ, JEN

The jump on E equal to zero and the jump on E not equal to zero check the status of the E register for parsing routines and user defined flags. In parse routines the E flag will be set to 1 if the token searched for is found, and Ø if not found. After returning from a parse routine it is convenient to set an error message or to do another procedure if the token is not found. Also, if the E register is used as a programming flag, it may be set on a special condition to jump to a procedure.

Example: To demand a numeric parameter at parse time:

JSB=NUMVA+	Try to parse a numeric value.
JEZ ERR	Jump if not found to error reporting.

JLZ, JLN

JLZ: Jump on left digit Ø (left BCD digit).
JLN: Jump on left digit not Ø.

Example: If R2Ø contains Ø11, the following code would take the jump:

```
TSB R2Ø
JLZ TRUE
```

JRZ, JRN

JRZ: Jump on right digit Ø (right BCD digit).
JRN: Jump on right digit nonzero.

Example: If Ø11 is in R2Ø, the following code would not take the jump:

```
TSB R2Ø
JRZ TRUE
```

ARP and DRP Load Instructions

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

ARP

Sets the address register pointer to the address register.

DRP

Sets data register pointer to the data register.

Use of R*

When entering source code, you may substitute R* for the AR or the DR in any CPU instruction. This causes the DRP or the DRP to be loaded with the least significant six bits of CPU register RØ. The effect is that the DR and the AR are specified by the contents of RØ. The CPU uses the DRP1 and ARP1 opcodes to implement this feature.

Example:

```
LDB RØ, = 26          Loads RØ with 26.
LDB R*,R3Ø            Loads CPU register specified by RØ.
                      (which is now R26) with the contents
                      of R3Ø.
STB R4Ø,R*            Stores contents of R4Ø into register
                      (R26 now) specified by RØ.
```

To avoid confusion, R1 is not allowed in either the DR or the AR fields. This means that CPU register R1 is inaccessible except by a multi-byte RØ operation or when RØ=1 and the ARP or the DRP is specified by R*.

Other Instructions

There are a few other CPU instructions which you can use.

BCD

Sets decimal mode (DCM=1). Arithmetic operations will be in BCD format.

BIN

Sets binary mode (DCM=0). Arithmetic operations will be in binary format.

CLE

The four bits of the extend register are set to 0.

DCE

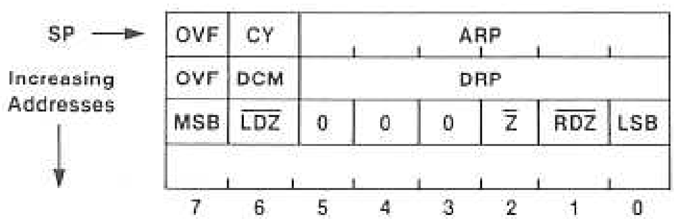
The extend register is decremented by 1. This instruction is always a binary operation, regardless of the DCM flag status.

PAD

Restores ARP, DRP, and status (usually after a SAD instruction) by popping them off the stack. The stack pointer is decremented by three, 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 the least LSB in bit 0, the $\overline{RDZ}$ bit 1, $\overline{Z}$ in bit 2, $\overline{LDZ}$ in bit 6, and MSB bit 7. The second byte is read as the DRP in bits 0-5, DCM status in bit 6, and overflow flag in bit 7. The third byte is read as the 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:



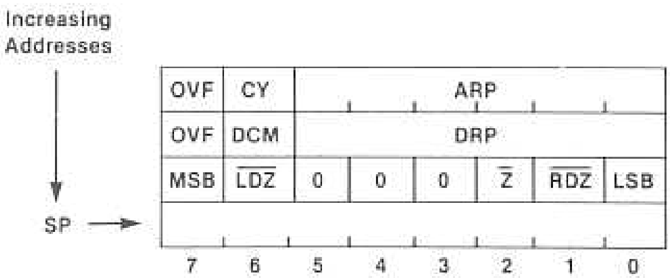
RTN

The subroutine return stack pointer is decremented by two. Then the return address is read from the stack and written into the program counter.

SAD

Three bytes are pushed onto the stack to save the ARP, the DRP, and the status flags (except E). The first byte contains the ARP in bits Ø-5, CY in bit 6, and the overflow flag in bit 7. The second byte contains the DRP in bits Ø-5, DCM status in bit 6, and the overflow flag in bit 7. The third byte contains the LSB in bit Ø, RDZ in bit 1, $\overline{Z}$ in bit 2, $\overline{LDZ}$ in bit 6, and the MSB in bit 7. The stack pointer is incremented by three. Status is not affected by this operation.

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



6.4 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.

A stack push instruction such as PUBD would be assembled and appear as shown here:

Byte Number	Machine Code	Source Code
ØØØ227	11Ø ØØ6 342	PUBD R1Ø,-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 (such as, 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 a DR, an AR, and a label, as in the case of indexed direct and indexed indirect addressing (such as, 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.

The ARP and the DRP During Assembly

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

If an instruction if not labeled (that is, 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.

Example:

Byte Number	Machine Code	Source Code
ØØØ227	11Ø ØØ6 342	LABEL POBD R1Ø,-R6
ØØØ232	342	POBD R1Ø,-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 R1Ø,-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. Times when they are not 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 subroutine jump, then returning, the first ARP and DRP encountered are not deleted.
- PAD: Following a PAD instruction, the first ARP and DRP are not deleted.

Pseudo-Instructions

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

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

- Assembly control.
- Data definition.
- Conditional assembly.

ABS base address

If the base address is less than 100000 (octal) then a ROM file will be generated at assembly time. Otherwise a binary program file will be generated and all labels are given absolute addresses, not relative addresses. The ABS statement must precede a NAM statement, if used.

FIN

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

GLO file name

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

LNK file name

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

LST

Causes the code to be listed on the current PRINTER IS device at assembly. The printed lines will be truncated at whatever the current line length is.

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 binary program #, unquoted string

Sets up the program control block for a binary program. Should be preceded only by GLO, ABS, LST, UNL, DAD, EQU, or comments.

ORG address

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

UNL

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 numeric value, unquoted string
ASC quoted string

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

ASP numeric value, unquoted string
ASP quoted string

Same as ASC except that the parity bit of the string's final character is set. (During operation, the 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 has been set, the system assumes the next character begins a new string or entry in the table.)

BSZ numeric value

Inserts into the object code the octal number of bytes of Ø's specified by the numeric value.

BYT numeric value [,numeric value...]

Inserts literal values into the object code.

DAD Label DAD address

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 DADs.

DEF label

Inserts the two-byte address associated with the label.

EQU Label EQU numeric value

Assigns either an absolute address or a constant to a label. This instruction is similar to the DAD pseudo-instruction.

GTO label

Generates four bytes of object code which load the program counter with the address, minus one, of the label. The label must be an absolute address.

The CPU relative jump instructions can cause jumps of from 177 (octal) to -2ØØ (octal) memory locations. The GTO pseudo-instruction is useful for jumping beyond this range.

The GTO 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.

VAL label

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

Pseudo-Instructions for Conditional Assembly

These instructions permit you to control assembly with conditional assembly flags. A conditional assembly flag has the same constraints as a label--it can be no more than eight characters in length, and the first character cannot be a digit.

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

AIF assembly flag name

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

CLR flag name

Changes the specified conditional assembly flag to false.

EIF

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 flag name

Sets the specified conditional assembly flag to true.

6.5 Multiple Binary Programs

There can be up to five binary programs in memory at one time. There is a table of two-byte addresses called BINBAS that contains the base addresses in the order in which the binary programs were loaded. Bytes that are not used are zero. Anytime the system calls a binary program, it first fetches from BINBAS the base address for that program and stores it in BINTAB.

The ASCII keyword tables and the binary programs are searched in the order they are loaded. This is also how initialization routines are called.

SAMPLE BINARY PROGRAMS

7.1 Introduction

This section includes five binary programs. In addition to being listed here, these programs are on the disc you received with your Assembler ROM. Source code file names end in "S", while object code file names end in "B."

Each of these programs is designed to illustrate assembly language programming, and each provides a function or keyword that is useful to the HP-87 operating system.

At the end of each program listing is a table of system routine addresses used by the program. Inserting the disc and placing a GLO GLOBAL pseudo-opcode near the beginning of the program eliminates the need for these addresses in some of the sample programs. Certain programs call system routines whose addresses are not available on the global file disc.

The string highlight program includes instruction on how to use a binary program following the listing.

7.2 String Highlight

Source Code: HGL$S

Object Code: HGL$B

```
1000 !***********************************************************************
1010 !*   This binary program implements a string function called HGL$      *
1020 !* which accepts one string parameter and returns that string with    *
1030 !* the most significant bit of each character set.                     *
1040 !*   This binary program is a translation of the UDL$ binary program *
1050 !* from the HP-85 Assembler Rom manual.                                 *
1060 !*                                                                     *
1070 !*                (c) Hewlett-Packard Co. 1982                          *
1080 !*                                                                     *
1090 !* An example of how this function might be used is:                  *
1100 !*                                                                     *
1110 !*    100 INPUT A$                                                     *
1120 !*    110 DISP HGL$(A$)                                                *
1130 !*                                                                     *
1140 !***********************************************************************
1150           NAM 53,HGLBIN          ! SET UP THE PROGRAM CONTROL BLOCK
1160           DEF RUNTIM             ! PTR TO RUNTIME ADDRESS TABLE
1170           DEF ASCIIS             ! PTR TO KEYWORD TABLE
1180           DEF PARSE              ! PTR TO PARSE ADDRESS TABLE
1190           DEF ERMSG              ! PTR TO ERROR MESSAGE TABLE
1200           DEF INIT               ! PTR TO INIT ROUTINE
1210 !***********************************************************************
1220 PARSE     BSZ 0                  ! NO PARSE ROUTINES
1230 !***********************************************************************
1240 RUNTIM    BYT 0,0                ! DUMMY TOK# 0 RUNTIME ADDRESS
1250           DEF REV.               ! TOK# 1 RUNTIME ADDRESS
1260           DEF HGL$.              ! TOK# 2 RUNTIME ADDRESS
1270           BYT 377,377            ! TERMINATE RELOCATION
1280 !***********************************************************************
1290 ASCIIS    ASP "HGL$B"            ! KEYWORD #1
1300           ASP "HGL$"             ! KEYWORD #2
1310 ERMSG     BYT 377                ! TERMINATE ASCII TABLE & ERMSG TABLE
1320 !***********************************************************************
1330 INIT      RTN                    ! NO INITIALIZATION TO BE DONE
1340 !***********************************************************************
1350           BYT 30,56              ! ATTRIBUTES ($ FUNCTION, 1 $ PARAMETER)
1360 HGL$.     POMD R45,-R12          ! POP STRING ADDRESS OFF OF STACK
1370           POMD R30,-R12          ! GET LENGTH OF STRING OFF OF STACK
1380           STM R30,R55            ! LENGTH NEEDS TO BE IN 55 FOR 'RSMEM-'
1390           CLB R57                ! ZERO OUT MSB OF RESERVED LENGTH
1400           JSB =RSMEM-            ! GO GET SOME TEMPORARY MEMORY
1410           PUMD R30,+R12          ! PUSH LENGTH BACK ONTO THE R12 STACK
1420           PUMD R65,+R12          ! PUSH ADDRESS RETURNED BY RSMEM ON STACK
1430           BIN                    ! MAKE SURE OF MATH MODE FOR LOOP COUNTER
1440           LDMD R75,=PTR1-        ! SAVE VALUE OF PTR1
1450           PUMD R75,+R6           !    ON R6 STACK
1460           LDB R34,=200           ! SET UP MASK
1470           STMD R45,=PTR1-        ! ADDRESS OF 1st BYTE OF ORIGINAL STRING
1480           STMD R65,=PTR2-        ! ADDRESS OF 1st BYTE OF RESERVED MEMORY
1490 MORE      DCM R30                ! DECREMENT LOOP COUNTER
1500           JNC DONE               ! JIF NO CHARACTERS LEFT
1510           LDBI R20,=PTR1-        ! GET NEXT CHARACTER FROM ORIGINAL STRING
1520           ORB R20,R34            ! SET MSB OF CURRENT CHARACTER
1530           STBI R20,=PTR2-        ! STORE HI-LIGHTED BYTE TO RESERVED MEMORY
1540           JMP MORE               ! GO GET SOME MORE
1550 DONE      POMD R75,-R6           ! RETRIEVE OLD VALUE OF PTR1
```

```
1560          STMD R75,=PTR1-          !   AND RESTORE IT BEFORE RETURNING
1570          RTN                      ! DONE
1580 !*****************************************************************
1590          BYT 0,56                 ! NO PARAMETERS, STRING FUNCTION
1600 REV.     BIN                      ! FOR ADDRESS MATH
1610          LDM R43,=40D,0           ! LOAD THE LENGTH OF THE STRING
1620          DEF DATE-                !    AND THE ADDRESS OF THE STRING
1630          BYT 0                    !         (MAKE IT A THREE BYTE ADDRESS)
1640          ADMD R45,=BINTAB         ! MAKE THE ADDRESS ABSOLUTE
1650          PUMD R43,+R12            ! PUSH IT TO THE OPERATING STACK
1660          RTN                      ! DONE
1670 DATE     ASC "40.102:veR  2891 ,oC drakcaP-ttelweH )C("
1680 DATE-    BSZ 0                    ! PLACE HOLDER FOR ADDRESS LOAD
1690 !*****************************************************************
1700 BINTAB   DAD 104070               !
1710 RSMEM-   DAD 31741                !
1720 PTR1-    DAD 177711               ! DEFINE ADDRESSES
1730 PTR2-    DAD 177715               !
1740          FIN                      ! TERMINATE ASSEMBLY
```

1. In assembler mode load the source code:

   ALOAD "HGL$S" [END LINE]

2. To assemble the source code:

   ASSEMBLE "HGL$B" [END LINE]

   If you want a printed copy of the object code as it assembles, you must designate a PRINTER IS device (that is, PRINTER IS 7Ø1). There must also be an LST instruction at the beginning of the code. The object code is now assembled and stored on your disc.

3. To use this function, return to BASIC mode. Type:

   BASIC [END LINE]

4. Load the object code. Type:

   LOADBIN "HGL$B" [END LINE]

5. Before running this program you may wish to set a breakpoint. With the system monitor inserted, type:

   BKP REL (1ØØ)

   The REL instruction sets the breakpoint at an absolute address in memory. The breakpoint information will appear on the CRT. It will also be printed if you specify a PRINTER IS device. For example:

   BKP REL (1ØØ),7Ø1

   The program will now halt at the address specified in the breakpoint. Your breakpoint will look similar to this when HGL$ ("string") is typed:

```
  PC    DR AR OV CY NG LZ ZP RZ OD DC  E    BKP1   BKP2    PTR1    PTR2   ROM
114334 57 55  0  0  0  1  0  0  1  0  00  114333 000000 0370014 0370013 000

       0   1   2   3   4   5   6   7     MEM 114233:0
R00  005 001 242 053 334 230 100 204     110 107 114 102 254 000 002 053  HGLB,   +
R10  307 200 350 212 075 210 001 001     110 107 114 044 102 040 040 040  HGL$B
R20  233 230 053 016 010 013 310 200     040 040 000 000 000 000 000 000
R30  041 000 324 230 267 230 233 230     233 230 277 230 307 230 277 230    ? G ?
R40  015 000 000 000 000 060 360 001     320 230 321 230 233 230 017 231  P Q
R50  110 053 230 002 000 041 000 001     324 230 377 377 110 107 114 044  T ■■HGL$
R60  000 000 000 000 000 041 350 001     302 110 107 114 244 377 236 030  BHGL$■
R70  016 004 000 000 000 017 360 001     056 145 012 343 130 343 055 243  .e cXc-#
```

After execution is halted at the breakpoint, you may single step a specified number of instructions using the TRACE instruction. For example, to trace the next 1Ø program steps, type:

TRACE 1Ø

The TRACE instruction will give you status information for each of those 1Ø steps, as well as the contents of memory. TRACE 1Ø will output the following information:

```
  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
114335 57 55  0  0  0  1  1  1  0  0  00  114334 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031741 57 55  0  0  0  1  1  1  0  0  00  114335 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031742 57 55  0  0  0  1  1  1  0  0  00  031741 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031743 57 55  0  0  0  1  1  1  0  0  00  031742 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031744 65 55  0  0  0  1  1  1  0  0  00  031743 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031747 65 55  0  0  0  1  0  0  1  0  00  031744 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031750 65 55  0  0  0  1  0  0  1  0  00  031747 000000 0370014 0370013 000

  PC    DR AR OV CY NG LZ ZR RZ OD DC  E    BKP1   BKP2    PTR1     PTR2   ROM
031751 65 55  0  0  0  1  0  0  0  0  00  114333 000000 0370014 0370013 000

       0   1   2   3   4   5   6   7     MEM 114233:0
R00   005 001 242 053 351 063 102 204    110 107 114 102 254 000 002 053   HGLB,  +
R10   307 200 350 212 075 210 001 001    110 107 114 044 102 040 040 040   HGL$B
R20   233 230 053 016 010 013 310 200    040 040 000 000 000 000 000 000
R30   041 000 324 230 267 230 233 230    233 230 277 230 307 230 277 230     ? G ?
R40   015 000 000 000 000 060 360 001    320 230 321 230 233 230 017 231   P Q
R50   110 053 230 002 000 041 000 000    324 230 377 377 110 107 114 044   T ■■HGL$
R60   000 000 000 000 000 256 231 000    302 110 107 114 244 377 236 030   BHGL$■
R70   016 004 000 000 000 017 360 001    056 145 012 343 130 343 055 243   .e cXc-#
```

To continue execution after a breakpoint or after a TRACE instruction, press [RUN].

6. To run this program without halting, type:

   HGL$ ("string")

   after the LOADBIN instruction.

7.3 CRT Control

Source Code: ALPHAS

Object Code: ALPHAB

```
1000 !*********************************************************************
1010 !* This binary program implements three CRT control statements:      *
1020 !*      AWRITE [<row>,<column>][,<string>]                            *
1030 !*      AREAD <string variable>                                       *
1040 !*      START CRT AT <absolute line #>                                *
1050 !* AWRITE allows you to do one of three things:                       *
1060 !*         1) force ALPHA mode without moving the cursor position     *
1070 !*         2) force ALPHA mode and move the cursor to a position which*
1080 !*            is relative to the top left of the current screen       *
1090 !*         3  force ALPHA mode and move the cursor to new position    *
1100 !*            and output a string at that location, leaving the cursor*
1110 !*            positioned at the beginning of the string.              *
1120 !*      In all cases the cursor is not actually displayed, until some *
1130 !*      other normal cursor movement occurs.                          *
1140 !* AREAD allows you to read a string of characters from the CRT into *
1150 !*      a string variable. Usually the cursor will have been moved to *
1160 !*      the correct position with the AWRITE statement.               *
1170 !* START CRT AT allows you to scroll the display up or down or jump  *
1180 !*      to an entirely different page, all under program control.     *
1190 !*      NOTE: this routine does not change the cursor's location in   *
1200 !*      CRT memory, so the cursor may get lost off of the screen when *
1210 !*      this command is used. It can be brought back by use of the    *
1220 !*      AWRITE statement, or by using the Home Cursor key.            *
1230 !* ALPHAB returns the revision date of the binary program.            *
1240 !*********************************************************************
1250 !*                                                                   *
1260 !* An example of how this binary might be used in BASIC is:           *
1270 !*     110 FOR I=1 TO 1000                                            *
1280 !*     120 START CRT AT IP(RND*50)                                    *
1290 !*     130 AWRITE RND*16,RND*80 @ AREAD A$                            *
1300 !*     140 AWRITE RND*16,RND*80,A$                                    *
1310 !*     150 NEXT I                                                     *
1320 !* This is guaranteed to turn any intelligent display into nonsense. *
1330 !*                                                                   *
1340 !*********************************************************************
1350 MYBPGM#   EQU 52                   ! BINARY PROGRAM NUMBER
1360           NAM 52,ALFA              ! NAME BLOCK FOR BINARY
1370           DEF RUNTIM               ! ADDRESS OF RUNTIME ADDRESSES
1380           DEF ASCIIS               ! ADDRESS OF ASCII TABLE
1390           DEF PARSE                ! ADDRESS OF PARSE ADDRESSES
1400           DEF ERMSG                ! ADDRESS OF ERROR MESSAGES
1410           DEF INIT                 ! ADDRESS OF INITIALIZATION ROUTINE
1420 RUNTIM    BSZ 2                    ! PLACE HOLDER
1430           DEF ALFA.                ! RUNTIME LABEL FOR 'AWRITE'
1440           DEF AREAD.               ! RUNTIME FOR 'AREAD'
1450           DEF STARTAT.             ! CRT TOP LINE
1460           DEF REV.                 ! RUNTIME FOR REVISION
```

```
1470 PARSE    BSZ 2                    ! PLACE HOLDER
1480          DEF ALPHAP               ! PARSE LABEL FOR 'AWRITE'
1490          DEF AREADP               ! PARSE LABEL FOR 'AREAD'
1500          DEF STARTATP             ! PARSE FOR TOP LINE
1510          BYT 377,377              ! END OF RELOCATABLES
1520 !******************************************************************
1530 ASCIIS   BSZ 0
1540          ASP "AWRITE"             ! TOKEN 1
1550          ASP "AREAD"              ! TOKEN 2
1560          ASP "START CRT AT"       ! TOKEN 3
1570          ASP "ALPHAB"             ! TOKEN 4
1580 ERMSG    BYT 377                  ! END OF ASCII TABLE
1590 !******************************************************************
1600 INIT     RTN                      ! NO INITIALIZATION TO BE DONE
1610 !******************************************************************
1620 STARTATP PUBD R43,+R6             ! SAVE TOKEN#
1630          JSB =NUMVA+              ! TRY TO GET A NUMBER
1640          JEZ ERR88                ! GOT AN ERROR
1650 OKAY     LDB R53,=371             ! BPGM TOKEN
1660          STBI R53,=PTR2-          ! STORE IT
1670          LDB R53,=MYBPGM#         ! GET MY BINARY NUMBER
1680          STBI R53,=PTR2-          ! STORE IT
1690          POBD R53,-R6             ! GET THE TOKEN NUMBER
1700          STBI R53,=PTR2-          ! STORE IT
1710          RTN                      ! ALL DONE
1720 !******************************************************************
1730 ALPHAP   PUBD R43,+R6             ! SAVE TOKEN NUMBER
1740          JSB =NUMVA+              ! TRY TO GET A NUMBER
1750          JEZ OKAY                 ! MUST BE JUST 'AWRITE'
1760          JSB =GETCMA              ! DEMAND A COMMA
1770          JSB =NUMVAL              ! DEMAND A NUMBER
1780          JEN OKAY2                ! JIF BOTH NUMBERS THERE
1790 ERR88    POBD R43,-R6             ! CLEAN UP R6
1800          JSB =ERROR+              ! ERROR HANDLING ROUTINE
1810          BYT 88D                  ! ERROR NUMBER
1820 OKAY2    CMB R14,=54              ! MAKE SURE OF A COMMA
1830          JNZ OKAY                 ! JIF JUST 'AWRITE X,Y'
1840          JSB =STREX+              ! PARSES A STRING EXPRESSION
1850          JEZ ERR88                ! JIF NO STRING TO ERROR
1860          JMP OKAY                 ! OTHERWISE FINISH UP THE PARSING
1870 ! *******************************************************************
1880 AREADP   PUBD R43,+R6             ! SAVE THE TOKEN
1890          JSB =SCAN                ! LET'S DO A SCAN
1900          JSB =STRREF              ! MUST BE A STRING REFERENCE
1910          JMP OKAY                 ! FINISH THE PARSE
1920 ! *******************************************************************
1930          BYT 0,56                 ! NO PARAMETERS, STRING FUNCTION
1940 REV.     BIN                      ! FOR ADMD R45,=BINTAB
1950          LDM R43,=40D,0           ! LOAD THE LENGTH OF THE STRING
1960          DEF DATE                 !      AND THE ADDRESS OF THE STRING
1970          BYT 0                    !          (MUST BE THREE BYTE ADDRESS)
1980          ADMD R45,=BINTAB         ! MAKE THE ADDRESS ABSOLUTE
1990          PUMD R43,+R12            ! PUSH IT ALL ON THE OPERATING STACK
2000          RTN                      ! DONE
2010          ASC "30.102:veR  2891 .oC drakcaP-ttelweH )c("
2020 DATE     BSZ 0                    ! PLACE HOLDER FOR THE LABEL (ADDRESS)
2030 ! *******************************************************************
```

```
2040          BYT 241                 ! BASIC STATEMENT
2050 ALFA.    BIN                     ! FOR MATH
2060          LDBD R37,=CRTSTS        ! CHECK CRT STATUS
2070          JPS INALPHA!            ! JIF ALREADY IN ALPHA MODE
2080          JSB =ALPHA.             ! IF NOT, MAKE IT SO
2090 INALPHA! CMMD R12,=TOS           ! ANYTHING ON THE R12 STACK
2100          JZR NO-ADR              ! JIF JUST 'AWRITE'
2110          JSB =DECUR2             ! KILL BOTH POSSIBLE CURSORS
2120          JSB =HMCURS             ! MOVE THE CURSOR TO THE HOME POSITION
2130          LDMD R14,=BINTAB        ! BECAUSE I'M RELATIVE
2140          CLM R43                 ! FAKE O STRING LENGTH
2150          LDM R20,R12             ! COPY OF R12
2160          SBM R20,=25,0           ! SUBTRACT 25
2170          CMMD R20,=TOS           ! WHAT'S ON R12
2180          JNZ A-ONLY              ! JIF ONLY X,Y
2190          POMD R43,-R12           ! GET LENGTH AND ADDRESS OF STRING
2200 A-ONLY   STMD R43,X14,SAV-$      ! SAVE LENGTH AND ADDRESS
2210          JSB =TWOB               ! GET TWO BINARY NUMBERS OFF OF R12
2220 CALCADR  DCM R56                 ! DECREMENT 'Y'
2230          JNG GOT-IT              ! JIF ADDRESS FIGURED OUT
2240          ADM R46,=120,0          ! ADD TO GET TO NEXT LINE
2250          JMP CALCADR             ! TRY FOR ANOTHER ONE
2260 GOT-IT   STM R46,R24             ! COPY ADDRESS DISPLACEMENT TO 26
2270          JSB =MOVCRS             ! MOVE THE CURSOR
2280          LDMD R43,X14,SAV-$      ! GET LENGTH AND ADDRESS OF STRING BACK
2290          LDM R56,R43             ! GET LENGTH
2300          JZR NO-ADR              ! JIF NO LENGTH
2310          STMD R45,=PTR2          ! SET MEMORY POINTER TO STRING ADDRESS
2320          LDM R36,R43             ! GET LENGTH
2330 ALOP     LDBI R32,=PTR2-         ! GET A CHARACTER
2340          JSB =CHKSTS             ! WAIT FOR CRT NOT BUSY
2350          STBD R32,=CRTDAT        ! STORE IT
2360          DCM R36                 ! ANY CHARACTERS LEFT
2370          JNZ ALOP                ! JIF THERE ARE
2380 NO-ADR   RTN                     ! ALL DONE
2390 ! ***********************************************************************
2400          BYT 241                 ! BASIC STATMENT
2410 AREAD.   BIN                     ! FOR MATH
2420          LDBD R37,=CRTSTS        ! GET CRT STATUS
2430          JPS INALPHA#            ! JIF ALREADY IN ALPHA MODE
2440          JSB =ALPHA.             ! IF NOT, MAKE IT SO
2450 INALPHA# JSB =DECUR2             ! KILL THE CURSORS
2460          POMD R73,-R12           ! GET STRING STUFF
2470          STM R73,R55             ! COPY TO 55
2480          PUMD R73,+R12           ! PUSH THE STUFF BACK
2490          CLB R57                 ! CLEAR MSB
2500          JSB =RESMEM             ! LET'S GO RESEARVE SOME MEMORY
2510          STM R55,R73             ! COPY 55 TO 75
2520          STM R65,R75             ! COPY SINK ADDRESS
2530          STMD R65,=PTR2          ! SET MEMORY POINTER
2540          PUMD R73,+R12           ! PUSH STRING STUFF ONTO R12
2550          TSM R55                 ! HOW BIG CAN I GO
2560          JZR DO-STO              ! JIF O
2570          LDMD R34,=CRTBYT        ! GET CURRENT POSITION
2580          PUMD R34,+R6            ! SAVE IT
2590          JSB =BYTCR!             ! SET CURRENT POSITION
```

```
ALOOP     JSB =INCHR              ! GO GET A CHARACTER
          STBI R32,=PTR2-         ! STORE IT
          JSB =RTCUR.             ! MOVE 1 BYTE
          DCM R55                 ! ANY MORE
          JNZ ALOOP               ! JIF THERE ARE
          POMD R34,-R6            ! GET OLD CRTBYT BACK
          JSB =BYTCR!             ! SET CURRENT POSITION
DO-STO    JSB =STOST              ! SAVE IT AWAY
          RTN                     ! ALL DONE
! ***********************************************************************
! *    START CRT AT THE SPECIFIED NUMBER                                *
! ***********************************************************************
          BYT 241
STARTAT.  JSB =ONEB               ! GET A NUMBER OFF OF R12
          BCD                     ! FOR MATH
          LLM R#                  ! *16
          BIN                     ! FOR THE REST
          STM R#,R#               ! COPY IT
          LLM R#                  ! *32
          LLM R#                  ! *64
          ADM R#,R#               ! *80
          STM R#,R#               ! COPY TO 46
          LDMD R#,=ASIZE          ! GET ALPHA SIZE INTO 76
          DRP R46                 ! GET READY FOR 'MOD'
          JSB =MOD                ! MOD IT
          STM R#,R34              ! COPY RESULT TO 34 FOR 'SAD1'
          JSB =SAD1               ! SET CRT START ADDRESS
          RTN                     ! ALL DONE
!*********************************************************************
SAV-$     BSZ 5                   ! SAVE AREA FOR ALPHA
!*********************************************************************
NUMVA+    DAD 22403               !
GETCMA    DAD 23477               !
NUMVAL    DAD 22406               !
STREXP    DAD 23724               !
ERROR+    DAD 10220               !
PTR2-     DAD 177715              !
SCAN      DAD 21110               !
STRREF    DAD 24056               !
STREX+    DAD 23721               !
BINTAB    DAD 104070              !
CRTSTS    DAD 177702              !
ONEB      DAD 12153               !
PTR2      DAD 177714              !
CHKSTS    DAD 13204               !
CRTBAD    DAD 177701              !
CRTDAT    DAD 177703              !
ALPHA.    DAD 12413               !
TOS       DAD 101744              ! DEFINE ADDRESSES
DECUR2    DAD 13467               !
HMCURS    DAD 13661               !
TWOB      DAD 56760               !
MOVCRS    DAD 13771               !
RESMEM    DAD 31741               !
CRTBYT    DAD 100206              !
BYTCR!    DAD 14003               !
INCHR     DAD 14262               !
```

```
3170 RTCUR.    DAD 13651            !
3180 STOST     DAD 46472            !
3190 ASIZE     DAD 104744           !
3200 SAD1      DAD 13723            !
3210 MOD       DAD 14216            !
3220           FIN                  ! TERMINATE ASSEMBLY
```

7.4 Line Input

Source Code: LINPUTS

Object Code: LINPUTB

```
!***********************************************************************
!*                                                                     *
!*        A KEYWORD THAT IS PARSED INTO MORE THAN ONE TOKEN            *
!*                                                                     *
!*        A TOKEN WITH A CLASS OF 44 (MISC IGNORE AT DECOMPILE)        *
!*         that makes it invisible when the program is listed          *
!*                                                                     *
!*                (c) 1982 Hewlett-Packard Co.                         *
!*                                                                     *
!*   This binary program implements the BASIC statement 'LINPUT'       *
!* which acts exactly the same as the BASIC statement 'INPUT' except *
!* that it will only allow you to input a string value and that        *
!* string value may contain commas and/or quotes. The keyword stands *
!* for Line INPUT, as it allows the inputing of a line regardless      *
!* of what characters are in that line.                                *
!*                                                                     *
!***********************************************************************
!*                                                                     *
!* An example of how a BASIC program might use LINPUT is:              *
!*                                                                     *
!*     100 DISP "Address of destination";                              *
!*     110 LINPUT Dest_addr$                                           *
!*     120 PRINT# 1; Dest_addr$                                        *
!*                                                                     *
!***********************************************************************
         NAM 51,LINPUT             ! SET UP PCB, BPGM # IS 51
         DEF RUNTIM                ! POINTER TO RUNTIME ADDRESS TABLE
         DEF ASCIIS                ! POINTER TO TABLE OF ASCII KEYWORDS
         DEF PARSE                 ! POINTER TO TABLE OF PARSE ADDRESSES
         DEF ERMSG                 ! POINTER TO TABLE OF ERROR MESSAGES
         DEF INIT                  ! POINTER TO INITIALIZATION ROUTINE
!***********************************************************************
!*  The way an INPUT statement works in the series 80 computers is    *
!* this: the keyword is actually parsed into two tokens, so the job   *
!* of doing an INPUT is split into three parts; two are performed by *
!* the two INPUT tokens and the third is performed by the system.     *
!* The first of the two tokens outputs a question mark to the CRT and*
!* puts the computer into a pseudo-calculator mode, which is known    *
!* as Idle-in-Input, by setting CSTAT (R16) to a 4, and then sets the*
!* immediate break bits in XCOM (R17) using "or"with 240(octal). Then*
!* the first token terminates its execution by returning to the       *
!* interpreter. The interpreter will see the immediate break bits in *
!* R17 and will drop out into the exec Loop. The exec will see that  *
!* the computer is in Idle-in-Input mode and will simply loop on      *
!* itself. At this point, the user starts typing his input (causing  *
!* keyboard interrupts, which set bits in R17 and SVCWRD, which cause*
!* the exec to call the character editor (CHEDIT), which echoes the  *
!* keys to the CRT, clears the SVCWRD flag, and returns to the exec).*
```

```
490 !* This continues until the END LINE key is pressed, which causes    *
500 !* CHEDIT to set a flag in the E register which will tell the exec   *
510 !* that END LINE has been pressed. This will cause the exec to resume*
520 !* execution of the BASIC program by re-entering the interpreter.    *
530 !* The third part of the INPUT is carried out by the second token of *
540 !* the INPUT statement. It takes the input line, parses and executes *
550 !* it, then stores the values in the appropriate variables.          *
560 !* LINPUT statement works in very much the same way. As a matter      *
570 !* of fact, the first two LINPUT tokens do nothing but call           *
580 !* the runtime code for the first of the INPUT tokens. The difference*
590 !* comes in the second token. For LINPUT, all we want to do is input *
600 !* a literal string with no expressions allowed, so we have no need  *
610 !* to parse and execute the input line. All we have to do is reverse *
620 !* the string so that the first character is at the highest address  *
630 !* and then store it in the string variable.                          *
640 !*******************************************************************
650 RUNTIM    BYT 0,0                  ! DUMMY ADDRESS FOR TOKEN #0 RUNTIME
660           DEF REV.                 ! ADDRESS FOR TOKEN #1 RUNTIME ROUTINE
670           DEF LINPT.               ! ADDRESS FOR TOKEN #2 RUNTIME ROUTINE
680           DEF LIN$.                ! ADDRESS FOR TOKEN #3 RUTNIME ROUTINE
690 !*******************************************************************
700 PARSE     BYT 0,0                  ! DUMMY ADDRESS FOR KEYWORD #0 PARSE
710           BYT 0,0                  ! DUMMY FOR KEYWORD #1 PARSE (A FUNCTION)
720           DEF LINPRS               ! ADDRESS FOR KEYWORD #2 PARSE ROUTINE
730           BYT 377,377              ! TERMINATE RELOCATION OF ADDRESSES
740 !*******************************************************************
750 !*  The runtime table has three entries even though the ASCII and    *
760 !* parse tables have only two. The third entry in the runtime table  *
770 !* will only be used in conjunction with the second entry. If you    *
780 !* look at the parse routine for the second keyword (LINPUT) you will*
790 !* see that it pushes out both tokens 2 and 3. Normally, you want to *
800 !* keep a one for one relationship between entries in the ASCII,     *
810 !* PARSE, and RUNTIME tables, but there are times when you can play  *
820 !* tricks like this (if you're careful).                             *
830 !*******************************************************************
840 ASCIIS    ASP "LINPUTG"            ! KEYWORD #1 (REVISION DATE FUNCTION)
850           ASP "LINPUT"             ! KEYWORD #2
860 ERMSG     BYT 377                  ! TERMINATE ASCII AND ERROR MESSAGE TABLES
870 !*******************************************************************
880 ERR89     JSB =ERROR+              ! SET ERROR FLAGS IN R17 AND 'ERRORS'
890           BYT 89D                  ! SYSTEM ERROR MESSAGE #89 'INVALID PARAM'
900 !*******************************************************************
910 LINPRS    LDM R55,=2,51,371        ! LOAD TOKEN#, BPGM#, AND SYSTEM TOKEN
920           STMI R55,=PTR2-          ! STORE THEM ALL OUT TO PARSE STREAM
930           JSB =SCAN                ! SCAN THE INPUT STREAM FOR NEXT TOKEN
940           JSB =STRREF              ! TRY TO GET A STRING VARIABLE REFERENCE
950           JEZ ERR89                ! JIF NOT THERE, ERROR CONDITION
960           LDM R55,=3,51,371        ! ELSE LOAD SECOND TOKEN#, BPGM#, AND SYS
970           STMI R55,=PTR2-          ! STORE THEM OUT TO PARSE OUTPUT STREAM
980 INIT      RTN                      ! DONE FOR PARSING AND INITIALIZING
990 !*******************************************************************
1000 !*  LINPT. is the runtime code for the first of the two LINUT tokens.*
1010 !* It is responsible for the output of the question mark to the CRT  *
1020 !* and putting the computer into Idle-in-Input mode.                 *
1030 !*******************************************************************
1040           BYT 241                 ! ATTRIB.,BASIC STATEMENT LEGAL AFTER THEN
1050 LINPT.    JSB =INPUT.             ! DO QUESTION MARK AND SET R16=4
1060           RTN                     ! DONE, WAIT FOR INPUT
```

```
!***********************************************************************
!*  LIN$. is the runtime code for the second of the the two LINUT     *
!* tokens. It is responsible for reversing the string in memory so it*
!* will be ready for storing into the string variable, and then doing*
!* the actual store (by calling STOST). The R12 stack will already    *
!* have been set up for the variable store by the tokens parsed by    *
!* STRREF.                                                            *
!***********************************************************************
          BYT 44               ! ATTRIBUTE, MISCELLANEOUS IGNORE
LIN$.     BIN                  ! BIN MODE FOR COUNTING
          LDMD R32,=INPTR      ! FETCH ADDRESS OF STRING THAT WAS INPUT
          STM R32,R14          ! SAVE A COPY
          CLM R36              ! PRE-SET LENGTH TO ZERO
CHRCNT    POBD R35,+R32        ! GET THE NEXT BYTE FROM INPUT STRING
          CMB R35,=15          ! IS IT A CARRIAGE RETURN CHARACTER?
          JZR ENDOF$           ! JIF YES, WE'VE FOUND THE END AND LENGTH
          ICM R36              !    ELSE INCREMENT THE LENGTH
          JMP CHRCNT           !    AND LOOP TO CHECK THE NEXT CHARACTER
ENDOF$    TSM R36              ! IS THE LENGHT ZERO?
          JZR DONE             ! JIF YES, RETURN A NULL STRING
          POBD R25,-R32        ! GET BACK TO LAST CHARACTER
POPBLK    POBD R25,-R32        ! FETCH LAST CHARACTER FROM END OF STRING
          CMB R25,=40          ! IS IT A BLANK?
          JNZ DONE+            ! JIF NO, CONTINUE ON
          DCM R36              !    ELSE DECREMENT LENGTH (TRIM BLANKS)
          JNZ POPBLK           ! JIF LENGTH NOT ZERO
DONE+     ICM R32              ! MOVE ADDRESS TO ONE HIGHER THAN END
          STM R32,R65          ! SET ADDRESS IN R65-R66
          CLB R67              ! CLEAR MOST SIGNIFICANT BYTE
DOLOOP    CMM R14,R32          ! FRONT OF STRING HIGHER OR EQUAL TO END?
          JCY DONE             ! JIF YES
          LDBD R30,R14         !    ELSE GET BYTE FROM FRONT
          POBD R31,-R32        !    AND A BYTE FROM THE BACK
          STBD R30,R32         !    STORE THE FRONT BYTE IN BACK
          PUBD R31,+R14        !    AND THE BACK BYTE IN FRONT
          JMP DOLOOP           ! LOOP TIL STRING IS REVERSED IN PLACE
DONE      PUMD R36,+R12        ! PUSH THE LENGTH OF STRING TO STACK
          PUMD R65,+R12        ! PUSH THE ADDRESS OF STRING TO STACK
          JSB =STOST           ! STORE THE STRING TO THE VARIABLE AREA
          RTN                  ! DONE
!***********************************************************************
!*  This is the runtime code for the revision date function, which is*
!* a string function with no parameters which always returns the same*
!* string value, the copyright notice and the revision code.          *
!***********************************************************************
```

```
1520          BYT 0,56              ! ATTRIBUTES,STRING FUNCTION,NO PARAMETERS
1530 REV.     LDM R43,=44,0         ! LOAD LENGTH OF THE STRING
1540          DEF DATE              ! AND THE ADDRESS OF THE STRING
1550          BYT 0                 !   (IT NEEDS TO BE A THREE BYTE ADDRESS)
1560          BIN                   ! BIN MODE FOR ADDRESS MATH
1570          ADMD R45,=BINTAB      ! MAKE THE ADDRESS ABSOLUTE
1580          PUMD R43,+R12         ! PUSH THE LENGTH AND ADDRESS TO THE STACK
1590          RTN                   ! DONE
1600          ASC "82.111 .veR 2891 drakcaP-ttelweH )c(" ! THE REVISION STRING
1610 DATE     BSZ 0                 ! NEED LABEL HERE TO GET RIGHT ADDRESS
1620 !*************************************************************************
1630 BINTAB   DAD 104070            !
1640 ERROR+   DAD 10220             !
1650 INPTR    DAD 101143            !
1660 INPUT.   DAD 16314             ! LABEL DEFINITIONS
1670 PTR2-    DAD 177715            !
1680 SCAN     DAD 21110             !
1690 STOST    DAD 46472             !
1700 STRREF   DAD 24056             !
1710          FIN                   ! TERMINATE ASSEMBLY
```

7.5 Taking the KYIDLE Hook and Buffering the Keyboard

Source Code: KEYS

Object Code: KEYB

```
!*********************************************************************
!*                                                                   *
!* TAKING THE 'KYIDLE' HOOK AND BUFFERING THE KEYBOARD               *
!*                                                                   *
!*              (c) 1981 Hewlett-Packard Co.                         *
!*                                                                   *
!*   THIS BINARY PROGRAM TAKES OVER THE 'KYIDLE' HOOK AND PUTS ALL   *
!* KEYS PRESSED INTO A BUFFER EXCEPT FOR THOSE KEYCODES LISTED IN    *
!* THE TABLE STARTING AT 'KEYTAB' (RIGHT NOW, THOSE KEYS TO BE LEFT  *
!* FOR THE SYSTEM TO HANDLE ARE THE SOFT KEYS AND THE RESET KEY. THIS*
!* COULD EASILY BE CHANGED BY MODIFYING THE 'KEYTAB' TABLE). THE     *
!* BINARY ALSO WATCHES FOR 'SHIFT END LINE' AND 'SHIFT UP ARROW'     *
!* (WHICH IS THE 'HOME' KEY. ('UP ARROW' AND 'HOME' ACTUALLY GENERATE*
!* THE SAME KEYCODE AND CAN ONLY BE DIFFERENTIATED BY CHECKING TO SEE*
!* IF THE SHIFT KEY IS UP OR DOWN.)) WHEN 'END LINE' OR 'UP ARROW' IS*
!* PRESSED WITH THE SHIFT KEY DOWN, THE BINARY PROGRAM CHANGES THE   *
!* KEYCODE TO A DIFFERENT UNIQUE KEYCODE SO THE BASIC PROGRAM CAN    *
!* TELL THE DIFFERENCE. THIS, AND SIMILAR TECHNIQUES, COULD BE       *
!* APPLIED TO MOST OF THE KEYBOARD.                                  *
!*                                                                   *
!*********************************************************************
!*                                                                   *
!* The following is a sample BASIC program showing how this binary   *
!* program can be used:                                              *
!*                                                                   *
!*   100 TAKE KEYBOARD                                               *
!*   110 A$=KEY$                                                     *
!*   120 IF A$="" THEN 110                                           *
!*   130 IF A$="E" THEN 200                                          *
!*   140 DISP "THAT WAS THE " & A$ & " KEY."                         *
!*   150 GOTO 110                                                    *
!*   200 RELEASE KEYBOARD                                            *
!*   210 DISP "DONE"                                                 *
!*   220 END                                                         *
!*                                                                   *
!*********************************************************************
MYBPGM#  EQU 50                ! BINARY PROGRAM NUMBER
         NAM 50,KEYS           ! NAME BLOCK FOR BINARY
         DEF RUNTIM            ! ADDRESS OF RUNTIME ADDRESSES
         DEF ASCIIS            ! ADDRESS OF ASCII TABLE
         DEF PARSE             ! ADDRESS OF PARSE ADDRESSES
         DEF ERMSG             ! ADDRESS OF ERROR MESSAGES
         DEF INIT              ! ADDRESS OF INITIALIZATION ROUTINE
RUNTIM   BSZ 2                 ! PLACE HOLDER
         DEF TAKE.             ! RUNTIME FOR 'TAKE KEYBOARD'
         DEF RELEAS.           ! RUNTIME FOR 'RELEASE KEYBOARD'
```

```
        DEF KEY$.               ! RUNTIME FOR 'KEY$'
        DEF REVDATE.            ! RUNTIME FOR REVISION
PARSE   BSZ 2                   ! PLACE HOLDER
        DEF COMPARS             ! PARSE ROUTINE FOR 'TAKE KEYBOARD'
        DEF COMPARS             ! PARSE ROUTINE FOR 'RELEASE KEYBOARD'
        BYT 377,377             ! END OF RELOCATABLES
!********************************************************************
ASCIIS  BSZ 0
        ASP "TAKE KEYBOARD"     ! TOKEN 1
        ASP "RELEASE KEYBOARD"  ! TOKEN 2
        ASP "KEY$"              ! TOKEN 3
        ASP "REV DATE"          ! TOKEN 4
ERMSG   BYT 377                 ! END OF ASCII TABLE
!********************************************************************
!*   BECAUSE THIS PROGRAM TAKES OVER 'KYIDLE', SOME SPECIAL TRICKS   *
!* ARE NEEDED. 'KYIDLE' IS AN INTERRUPT HOOK WHICH MEANS THAT THE    *
!* BASE ADDRESS OF THIS BINARY PROGRAM MAY NOT BE IN 'BINTAB'. A     *
!* METHOD IS NEEDED FOR THE HOOK ROUTINE ('USEKEY' IN THIS CASE) TO  *
!* KNOW WHAT THE BASE ADDRESS IS. SINCE THE 'KYIDLE' HOOK IS 7 BYTES *
!* LONG AND IT ONLY TAKES 4 BYTES TO DO 'JSB =USEKEY' & 'RTN', 3      *
!* BYTES ARE LEFT UNUSED (AND THAT WE CAN BE SURE NO ONE ELSE IS     *
!* GOING TO USE, AS LONG AS THIS BINARY HAS THE HOOK, WHICH IS AS    *
!* LONG AS IT MATTERS). TWO OF THESE BYTES ARE USED TO STORE THE     *
!* BASE ADDRESS OF THIS BINARY PROGRAM. WE'VE NAMED THE LOCATION     *
!* 'MYBTAB' AND DEFINED ITS ADDRESS AS 4 HIGHER THAN THAT OF 'KYIDLE'*
!* (103703 AND 103677 RESPECTIVELY.)                                 *
!*    THE 'INIT' ROUTINE DOESN'T HAVE TO DO ANYTHING IN THIS PROGRAM *
!* SINCE 'LOAD' AND 'SCRATCH' CAN'T BE PERFORMED WHILE THE BINARY    *
!* HAS THE HOOK, AND DURING A 'RESET' THE SYSTEM WILL HAVE ALREADY   *
!* PUT 'RTN's BACK INTO 'KYIDLE'. WE ONLY TAKE THE HOOK WHEN A       *
!* 'TAKE KEYBOARD' COMMAND IS EXECUTED, SO THERE'S NOTHING FOR INIT  *
!* TO DO.                                                            *
!*    THE BASIC PROGRAM WRITER NEEDS TO BE VERY CAREFUL, HOWEVER,    *
!* USING THIS BINARY, BECAUSE IF HE WERE TO EXECUTE A 'STOP' OR 'END'*
!* COMMAND WHILE THE HOOK IS TAKEN, THE KEYBOARD WILL EFFECTIVELY BE *
!* LOCKED UP EXCEPT FOR THE 'RESET' KEY AND, THUS, 'RESET' WOULD THEN*
!* BE THE USERS ONLY RECOURSE.                                       *
!********************************************************************
INIT    RTN                     ! ALL DONE
!********************************************************************
!* NEITHER 'TAKE KEYBOARD' OR 'RELEASE KEYBOARD' HAVE ANY PARAMETERS *
!* SO THEY BOTH USE THE SAME PARSE ROUTINE, WHICH SIMPLY PUSHES OUT  *
!* THE THREE BYTE SEQUENCE FOR THE KEYWORD AND THEN DOES A 'SCAN' FOR*
!* THE SYSTEM, SO THAT R14 WILL HAVE THE NEXT TOKEN WHEN WE RETURN.  *
!********************************************************************
COMPARS LDM R56,=50,371         ! BPGM # AND SYSTEM TOKEN
        LDB R55,R43             ! GET THE BINARY PROGRAM TOKEN #
        STMI R55,=PTR2-         ! STORE IT ALL OUT TO PARSE STACK
        JSB =SCAN               ! DO A SCAN FOR THE SYSTEM
        RTN
```

```
!**********************************************************************
!* 'REV DATE' IS A STRING FUNCTION WITH NO PARAMETERS WHICH RETURNS   *
!* AS ITS STRING VALUE THE COPYRIGHT STATEMENT AND REVISION CODE OF   *
!* THE BINARY PROGRAM.                                                *
!**********************************************************************
           BYT 0,56                ! NO PARAMETERS, STRING FUNCTION
REVDATE.   BIN                     ! FOR ADMD R45,=BINTAB
           LDM R43,=40D,0          ! LOAD THE LENGTH OF THE STRING
           DEF DATE                !       AND THE ADDRESS OF THE STRING
           BYT 0                   !          (MUST BE THREE BYTE ADDRESS)
           ADMD R45,=BINTAB        ! MAKE THE ADDRESS ABSOLUTE
           PUMD R43,+R12           ! PUSH IT ALL ON THE OPERATING STACK
           RTN                     ! DONE
           ASC "31.102:veR  .oC drakcaP-ttelweH 2891 )c("
DATE       BSZ 0                   ! PLACE HOLDER FOR THE LABEL (ADDRESS)
!**********************************************************************
!* THIS IS THE TABLE OF KEYS THAT THE BINARY PROGRAM SHOULD LET THE   *
!* SYSTEM HANDLE, AND IT SHOULD NOT PUT THEM IN THE BUFFER. THE TABLE*
!* IS TERMINATED BY A 377, WHICH IS A KEYCODE THE KEYBOARD CONTROLLER*
!* IC IS INCAPABLE OF GENERATING.                                     *
!**********************************************************************
KEYTAB     BYT 200                 ! K1
           BYT 201                 ! K2
           BYT 202                 ! K3
           BYT 203                 ! K4
           BYT 241                 ! K5
           BYT 242                 ! K6
           BYT 234                 ! K7
           BYT 204                 ! K8
           BYT 205                 ! K9
           BYT 206                 ! K10
           BYT 207                 ! K11
           BYT 245                 ! K12
           BYT 254                 ! K13
           BYT 223                 ! K14
           BYT 213                 ! RESET
           BYT 377                 ! END OF INVALID KEY TABLE
!**********************************************************************
!*   THIS IS THE RUNTIME ROUTINE FOR THE 'TAKE KEYBOARD' KEYWORD. IT *
!* INITIALIZES POINTERS TO THE BEGINNING AND END OF THE KEYBOARD      *
!* BUFFER, WHICH EXISTS FARTHER DOWN IN THE BINARY PROGRAM, TAKES     *
!* OVER THE 'KYIDLE' HOOK, AND INVALIDATES THE KEY REPEAT FLAG. IF    *
!* THE KEY REPEAT FLAG IS VALID, THE LAST KEY IS TAKEN FROM THE       *
!* BUFFER (USING THE 'KEY$' FUNCTION), AND A KEY IS STILL DEPRESSED  *
!* THE LAST KEY WILL BE PUT BACK IN THE BUFFER SO THAT IT WILL REPEAT*
!* AS LONG AS THE KEY IS HELD DOWN.                                   *
!**********************************************************************
           BYT 241
TAKE.      LDMD R46,=BINTAB        ! FOR RELATIVE ADDRESSING
           LDM R30,=KEYBUF         ! GET ADDRESS OF KEYBOARD BUFFER
           ADM R30,R46             ! MAKE IT ABSOLUTE
           STMD R30,X46,KEYPTR     ! INITIALIZE KEY POINTER
           ADM R30,=80D,0          ! POINT TO END OF BUFFER
```

```
2490          STMD R30,X46,KEYEND     ! INITIALIZE KEYEND
2500          LDM R30,=USEKEY         ! ADDRESS OF KEYBOARD SERVICE ROUTINE
2510          ADM R30,R46             ! MAKE IT ABSOLUTE
2520          STM R30,R43             ! COPY TO 43&44
2530          LDB R45,=236            ! 45='RTN'
2540          LDB R42,=316            ! 42='JSB'
2550 TAKEIT   STMD R#,=KYIDLE         ! STORE OUT RTN'S OR JSB=USEKEY,RTN,BINTAB
2560          LDB R#,=377             ! INVALID REPEAT FLAG
2570          STBD R#,X46,LASTKEY     ! SET IT
2580          RTN
2590 !*****************************************************************
2600 !* THIS IS THE RUNTIME ROUTINE FOR THE 'RELEASE KEYBOARD' KEYWORD.  *
2610 !* ALL IT DOES IS PLACE RETURNS BACK INTO THE 'KYIDLE' HOOK, THUS,  *
2620 !* GIVING UP CONTROL OF THE KEYBOARD.                               *
2630 !*****************************************************************
2640          BYT 241
2650 RELEAS.  LDMD R46,=BINTAB        ! GET BPGM'S BASE ADDRESS
2660          LDM R52,=236,236,236,236,236,236 ! LOTS OF RTNS
2670          JMP TAKEIT              ! GO STORE TO HOOK
2680 !*****************************************************************
2690 !*  'USEKEY' IS AN INTERRUPT SERVICE ROUTINE SO IT MUST BE CAREFUL TO*
2700 !* SAVE ALL CPU STATUS AND CONTENTS AND THEN RESTORE THEM WHEN DONE. *
2710 !* THE SYSTEM HAS ALREADY DONE A 'SAD' BEFORE IT DID THE 'JSB' TO    *
2720 !* 'KYIDLE'. THE ROUTINE CHECKS TO SEE IF THE BUFFER IS FULL AND IF  *
2730 !* SO THROWS THE CURRENT KEYHIT AWAY. IT THEN CHECKS FOR THE SHIFTED *
2740 !* 'UP ARROW' OR 'END LINE' KEYS AND IF SO MODIFIES THE KEYCOD TO    *
2750 !* MATCH. IT THEN CHECKS THE 'KEYTAB' TABLE TO SEE IF THIS KEY SHOULD*
2760 !* BE IGNORED. IF IT IS IN THE TABLE, THE ROUTINE JUST CLEANS UP A   *
2770 !* LITTLE AND RETURNS BACK INTO THE SYSTEM KEY HANDLING ROUTINE.     *
2780 !* OTHERWISE, IT PUTS THE NEW KEYCODE IN THE BUFFER AND UPDATES THE  *
2790 !* BUFFER POINTER. IT THEN FIGURES OUT WHAT THE DRP SHOULD BE WHEN   *
2800 !* IT RETURNS FROM THE INTERRUPT SERVICE, AND PLACES A DRP COMMAND   *
2810 !* WHERE IT WILL BE EXECUTED JUST BEFORE RETURNING (THIS IS SO THE   *
2820 !* EXTENDED MEMORY CONTROLLER CAN KEEP TRACK OF THE DRP FOR MULTI-   *
2830 !* BYTE OPERATIONS.) IT THEN RESTORES REGISTERS, THROWS AWAY TWO     *
2840 !* RETURN ADDRESSES, AND RETURNS TO WHATEVER WAS HAPPENING BEFORE    *
2850 !* THE KEYBOARD INTERRUPTED.                                         *
2860 !*****************************************************************
2870 USEKEY   STBD R#,=GINTDS         ! DISABLE GLOBAL INTERRUPTS
2880          BIN                     ! FOR EVERYTHING
2890          PUMD R2,+R6             ! SAVE 2&3
2900          PUMD R40,+R6            ! SAVE THE 40'S
2910          LDM R40,R20             ! AND THE 20'S
2920          LDMD R26,=MYBTAB        ! FOR RELATIVE ANYTHING
2930          LDMD R20,X26,KEYPTR     ! GET THE KEY POINTER
2940          LDMD R22,X26,KEYEND     ! ADDRESS OF END OF BUFFER
2950          CMM R22,R20             ! BUFFER FULL?
2960          JZR RE-START            ! JIF IT IS
2970          LDBD R22,=KEYCOD        ! GET THE KEY CODE
2980          LDBD R25,=KEYSTS        ! GET KEYBOARD STATUS
2990          ANM R25,=10             ! MASK FOR SHIFT KEY
3000          JZR NOTSHIFT            ! JIF SHIFT KEY NOT DOWN
3010          CMB R22,=kUPCUR         ! UP CURSOR KEY?
3020          JNZ ENDLINE?            ! JIF NOT
3030          LDB R22,=kHOME          ! OTHERWISE MAKE IT THE HOME KEY
3040          JMP NOTSHIFT            ! FALL THROUGH
3050 ENDLINE? CMB R22,=kENDLINE       ! WAS IT THE ENDLINE KEY?
3060          JNZ NOTSHIFT            ! JIF NOT
3070          LDB R22,=kSENDLIN       ! MAKE IT SHIFT ENDLINE
3080 NOTSHIFT LDM R24,=KEYTAB         ! ADDRESS OF INVALID KEYS
3090          ADM R24,R26             ! MAKE IT ABSOLUTE
3100 KEYLOOP  POBD R23,+R24           ! GET AN INVALID KEYCODE
3110          CMB R23,=377            ! END OF TABLE?
```

```
3120          JZR KEYLOOP1            ! JIF IT IS
3130          CMB R23,R22             ! IS THIS KEY INVALID
3140          JNZ KEYLOOP             ! JIF NO MATCH
3150          JSB X26,FIXUP-R6        ! FIX UP THE R6 STACK
3160          JMP KEYRTN+             ! FALL THROUGH, LET THE SYSTEM HAVE IT
3170 KEYLOOP1 PUBD R22,+R20           ! APPEND TO THE BUFFER
3180          STMD R20,X26,KEYPTR     ! UPDATE THE POINTER
3190 RE-START CLB R20                 ! \
3200          ICB R20                 !  > RESTART THE KEYBOARD SCANNER
3210          STBD R20,=KEYCOD        ! /
3220          JSB X26,FIXUP-R6        ! FIX UP THE R6 STACK
3230          SBM R6,=4,0             ! TRASH TWO RETURNS
3240 KEYRTN   STBD R#,=GINTEN         ! RE-ENABLE GLOBAL INTERRUPTS
3250 DRP      BSZ 1                   ! FORCE THE DRP
3260          PAD                     ! RESTORE  THE STATUS
3270          RTN                     ! ALL DONE
3280 KEYRTN+  STBD R#,=GINTEN         ! RE-ENABLE INTERRUPTS
3290          RTN
3300 FIXUP-R6 STMI R30,=MYBTAB        ! SAVE 30
3310          POMD R30,-R6            ! GET THE RETURN ADDRESS
3320          LDM R20,R6              ! COPY OF R6
3330          SBM R20,=20,0           ! GET DOWN TO MIDDLE OF THE SAD
3340          LDBD R20,R20            ! FETCH THE DRP BYTE
3350          ANM R20,=77,0           ! MASK OUT THE LAST DRP
3360          ADB R20,=100            ! MAKE IT INTO A DRP INSTRUCTION
3370          STBD R20,X26,DRP        ! STORE IT OUT
3380          STM R40,R20             ! RESTORE  THE 20'S
3390          POMD R40,-R6            ! RESTORE  THE 40'S
3400          POMD R2,-R6             ! RESTORE 2&3
3410          PUMD R30,+R6            ! PUT THE RETURN BACK
3420          LDMI R30,=MYBTAB        ! GET 30 BACK
3430          RTN                     ! ALL DONE
3440 !*******************************************************************
3450 !*   THIS IS THE RUNTIME ROUTINE FOR THE 'KEY$' KEYWORD. IT IS A      *
3460 !* STRING FUNCTION WITH NO PARAMETERS WHICH RETURNS A STRING WITH A  *
3470 !* LENGTH OF ONE WHOSE SOLE CHARACTER IS THE KEYCODE OF THE FIRST    *
3480 !* KEY IN THE KEYBOARD BUFFER. IF THE BUFFER WAS EMPTY, IT RETURNS   *
3490 !* A NULL STRING (LENGTH=0). WHEN IT TAKES A KEY OUT OF THE BUFFER,  *
3500 !* IT COLLAPSES ALL THE OTHER KEYCODES IN THE BUFFER AND ADJUSTS THE *
3510 !* BUFFER POINTER.                                                   *
3520 !*******************************************************************
3530          BYT 0,56
3540 KEY$.    BIN                     ! FOR EVERYTHING
3550          STBD R#,=GINTDS         ! DISABLE GLOBAL INTERRUPTS
3560          LDMD R14,=BINTAB        ! FOR ANYTHING RELATIVE
3570          LDM R20,=KEYBUF         ! ADDRESS OF KEYBOARD BUFFER
3580          ADM R20,R14             ! MAKE ADDRESS ABSOLUTE
3590          LDMD R22,X14,KEYPTR     ! GET POINTER INTO BUFFER
3600          CMM R22,R20             ! BUFFER EMPTY?
3610          JZR KEY$3               ! JIF IT IS
3620          LDM R30,R20             ! COPY 20
3630          POBD R32,+R20           ! GET A KEY
3640          STBD R32,X14,LASTKEY    ! SAVE LAST KEY FOR POSSIBLE REPEAT
```

```
3650 KEY$1     CMM R22,R20           ! BUFFER COLLAPSED
3660           JZR KEY$2             ! JIF IT IS
3670           POBD R33,+R20         ! GET A KEY
3680           PUBD R33,+R30         ! MOVE IT DOWN
3690           JMP KEY$1             ! LOOP
3700 KEY$2     DCM R22               ! ADJUST KEYPTR
3710           STMD R22,X14,KEYPTR   ! AND RESTORE  IT
3720 KEY$2+    CLM R22               ! \
3730           ICM R22               !  > LENGTH OF 1
3740 KEY$2++   PUMD R#,+R12          ! /
3750           LDM R55,=LASTKEY      ! ADDRESS OF KEYHIT
3760           BYT 0                 ! ---->  R57
3770           ADMD R55,=BINTAB      ! MAKE ADDRESS ABSOLUTE
3780           ICM R55               ! POINT TO AFTER THE KEY
3790           PUMD R55,+R12         ! PUSH ADDRESS OUT
3800           STBD R#,=GINTEN       ! RE-ENABLE GLOBAL INTERRUPTS
3810           RTN                   ! ALL DONE
3820 KEY$3     LDBD R32,X14,LASTKEY  ! CHECK LAST KEY
3830           CMB R32,=377          ! INVALID REPEAT?
3840           JZR KEY$4             ! JIF SO
3850           LDBD R32,=KEYSTS      ! GET KEYBOARD STATUS
3860           LRB R32               ! SHIFT STILL DOWN FLAG
3870           JOD KEY$2+            ! LET'S REPEAT IT
3880 KEY$4     LDB R32,=377          ! INVALID REPEAT FLAG
3890           STBD R32,X14,LASTKEY  ! SET INVALID REPEAT
3900           CLM R32               ! NO REPEAT, SO 0 LENGTH
3910           JMP KEY$2++           ! ONE MORE TIME
3920 ! ****************************************************************
3930 LASTKEY   BSZ 1                 ! FOR KEY REPEATING PURPOSES
3940 KEYBUF    BSZ 80D               ! ALLOW UP TO 80 KEY STROKES IN BUFFER
3950 KEYPTR    BSZ 2                 ! POINTER TO INPUT POINT IN BUFFER
3960 KEYEND    BSZ 2                 ! POINTER TO END OF THE BUFFER
3970 kUPCUR    EQU 243               ! UP CURSOR KEYCODE
3980 kHOME     EQU 230               ! NEW HOME KEYCODE
3990 kENDLINE  EQU 232               ! ENDLINE KEYCODE
4000 kSENDLIN  EQU 227               ! NEW SHIFT END LINE KEYCODE
4010 ! ****************************************************************
4020 ERROR+    DAD 10220             !
4030 PTR2-     DAD 177715            !
4040 SCAN      DAD 21110             !
4050 BINTAB    DAD 104070            !
4060 PTR2      DAD 177714            !
4070 ROMFL     DAD 104065            ! DEFINE SYSTEM ADDRESSES
4080 KYIDLE    DAD 103677            !
4090 GINTDS    DAD 177401            !
4100 GINTEN    DAD 177400            !
4110 MYBTAB    DAD 103703            !
4120 KEYCOD    DAD 177403            !
4130 KEYSTS    DAD 177402            !
4140           FIN                   ! TERMINATE ASSEMBLY
```

7.6 GET and SAVE

Source Code: GETSAVES

Object Code: GETSAVEB

```
!**********************************************************************
!*  This binary program implements the SAVE and GET statements for     *
!* turning programs into normal strings in a DATA file and turning     *
!* normal strings back into lines of a BASIC program.                  *
!*  The syntax for the two statements is:                              *
!*      SAVE <file name>[,<beginning line>][,<ending line>]            *
!*      GET <file name>                                                *
!*  SAVE calculates the size of the DATA file needed by listing the    *
!* program and counting the total length of the strings (plus the      *
!* three bytes of header per string required by the file manager).     *
!* It does this by taking over IOTRFC and forcing the select code to   *
!* a value that will cause the listed strings to go out through the    *
!* hook. LSSET is an entry point in the LIST routine that lists the    *
!* entire program. After the size of the data file is known, it is     *
!* created (any old one of that name already in existence will be      *
!* purged first) and then the program is listed again, this time with*
!* the lines (as strings) being printed out to the data file.          *
!*  GET opens the data file, reads a string, copies the string to      *
!* the input buffer INPBUF, then calls the PARSER, which will parse    *
!* the line and edit it into the program, if no errors occur. If a     *
!* parse error occurs, an exclamation point is inserted into the line*
!* after the line number and the line is parsed again as a comment.    *
!* GET has to create a dummy string variable area in the binary        *
!* program for the strings to be read into, because RDSTR. does a      *
!* call to STOST before it returns, and STOST expects all the usual    *
!* information on the stack and an associated variable area (in        *
!* other words, we have to trick the system when we call RDSTR.).      *
!**********************************************************************
         NAM 41,SAVG            ! SET UP THE PROGRAM CONTROL BLOCK
         DEF RUNTIM             ! PTR TO THE RUNTIME ADDRESSES
         DEF TOKS               ! PTR TO THE KEYWORDS
         DEF PARSE              ! PTR TO THE PARSE ADDRESSES
         DEF ERMSG              ! PTR TO THE ERROR MESSAGE TABLE
         DEF INIT               ! PTR TO THE INITIALIZATION ROUTINE
RUNTIM   BYT 0,0                ! DUMMY RUNTIME ADDRESS FOR TOK# 0
         DEF SAVE.              ! RUNTIME ADDRESS FOR TOK# 1
         DEF REVISON.           ! RUNTIME ADDRESS FOR TOK# 2
         DEF GET.               ! RUNTIME ADDRESS FOR TOK# 3
PARSE    BYT 0,0                ! DUMMY PARSE ADDRESS FOR TOK# 0
         DEF SAVPARS            ! PARSE ADDRESS FOR TOK# 1
         BYT 0,0                ! DUMMY PARSE ADDRESS FOR TOK# 3
         DEF GETPARS            ! PARSE ADDRESS FOR TOK# 3
ERMSG    BYT 377,377            ! TERMINATE RELOCATION AND ERROR TABLE
INIT     RTN                    ! NO INITIALIZATION
TOKS     ASP "SAVE"             ! KEYWORD #1
         ASP "GET SAVE"         ! KEYWORD #2
         ASP "GET"              ! KEYWORD #3
         BYT 377                ! TERMINATE KEYWORD TABLE
```

```
1480 !*********************************************************************
1490 SAVPARS  PUBD R43,+R6              ! SAVE CURRENT TOKEN
1500          JSB =STREX+               ! GET THE FILE NAME
1510          JEN OK1                   ! JIF IT WAS THERE
1520 ERR      POBD R43,-R6              ! ELSE CLEAN UP STACK
1530          JSB =ERROR+               ! REPORT THE ERROR
1540          BYT 88D                   ! BAD STATEMENT
1550 OK1      CMB R14,=54               ! COMMA?
1560          JNZ PARSCOMN              ! JIF NO LINE NUMBERS
1570          JSB =GO12N                ! ELSE GET ONE OR TWO LINE NUMBERS
1580          LDBI R56,=PTR2+           ! CLEAN UP PARSE STREAM
1590 PARSCOMN LDM R56,=41,371           ! BPGM# AND SYSTEM TOKEN
1600          POBD R55,-R6              ! RECOVER BPGM TOKEN #
1610          STMI R55,=PTR2-           ! STORE THEM OUT TO THE PARSE STREAM
1620          RTN                       ! DONE
1630 !*********************************************************************
1640 GETPARS  PUBD R43,+R6              ! SAVE THE INCOMING TOKEN
1650          JSB =STREX+               ! GET THE FILE NAME
1660          JEZ ERR                   ! JIF NOT THERE
1670          JMP PARSCOMN              ! ELSE FINISH UP
1680 !*********************************************************************
1690          BYT 241                   ! BASIC STATEMENT, LEGAL AFTER THEN
1700 SAVE.    JSB =CLEAR.               ! CLEAR THE CRT
1710          LDMD R10,=BINTAB          ! GET OUR BASE ADDRESS
1720          LDM R26,=SAVING           ! GET THE RELATIVE ADDRESS OF MSG
1730          ADM R26,R10               ! MAKE IT ABSOLUTE
1740          LDM R36,=20,0             ! LOAD THE LENGTH OF THE MSG
1750          JSB =OUTSTR               ! OUTPUT THE MSG
1760          LDMD R41,=IOTRFC          ! SAVE THE REAL HOOK CONTENTS
1770          STMD R41,X10,SAVIOTFC     ! STORE IT AWAY
1780          LDMD R40,=SCTEMP          ! SAVE THE REAL SELECT CODE
1790          STMD R40,X10,SAVSCTEM     ! STORE IT AWAY
1800          LDM R72,=231,231,11,0,0,0 ! LOAD DEFAULT LIST PARAMETERS
1810          STMD R72,=LLDCOM          ! SET THEM
1820          LDM R20,R12               ! COPY STACK POINTER
1830          SBM R20,=5,0              ! TAKE OFF STRING STUFF
1840          CMMD R20,=TOS             ! ANYTHING ELSE THERE?
1850          JZR DO-IT                 ! JIF NO, USE DEFAULTS
1860          JSB =ONEI                 ! ELSE GET ONE NUMBER OFF
1870          LDM R20,R12               ! COPY STACK POINTER
1880          SBM R20,=5,0              ! ADJUST FOR STRING STUFF
1890          CMMD R20,=TOS             ! ANY MORE?
1900          JZR STOLIN1               ! JIF NO
1910          STMD R45,=LLDCOM          ! ELSE SET LAST LINE DECOMPILE
1920          JSB =ONEI                 ! GET THE FIRST LINE
1930 STOLIN1  STMD R45,=FLDCOM          ! SET THE FIRST LINE DECOMPILE
1940 DO-IT    POMD R43,-R12             ! GET THE STRING
1950          STMD R43,X10,FILENAME     ! SAVE IT AWAY
1960          CLM R50                   ! SET UP FOR A FLOATING POINT 1
1970          LDB R57,=10C              ! THAT FINISHES IT
1980          PUMD R50,+R12             ! PUSH TO STACK FOR ASSIGN# 1 TO
1990          PUMD R43,+R12             ! PUSH FILE NAME BACK
2000          PUMD R10,+R6              ! SAVE OUR BASE ADDRESS
2010          JSB =ROMJSB               ! SELECT THE MSTORAGE ROM
2020          DEF ASSIG.                ! ASSIGN BUFFER # 1 TO FILE
2030          VAL MSROM#                ! ROM TO SELECT
2040          POMD R10,-R6              ! RECOVER OUR BASE
2050          CMB R17,=300              ! ANY ERRORS?
2060          JNC ITSTHERE              ! JIF NO, IT WAS THERE AND DATA FILE
2070          LDBD R20,=ERRORS          ! GET REASON
```

```
2080           CMB R20,=67D           ! FILE NAME ERROR?
2090           JZR CREATIT            ! JIF IT WASN'T THERE
2100           GTO RESTORE            ! ELSE BAIL OUT
2110 ITSTHERE  PUMD R10,+R6           ! SAVE OUR BASE
2120           STMD R12,=TOS          ! MAKE SURE STACK LOOKS GOOD
2130           LDMD R73,X10,FILENAME  ! GET THE FILE NAME BACK
2140           PUMD R73,+R12          ! PUSH IT TO THE STACK
2150           JSB =ROMJSB            ! SELECT THE ROM
2160           DEF MSPUR.             ! PURGE THE FILE
2170           VAL MSROM#             ! ROM TO SELECT
2180           POMD R10,-R6           ! RECOVER OUR BASE ADDRESS
2190           JMP CREATIT            ! CONTINUE
2200 CALCRTN   POMD R10,-R6           ! RECOVER BASE
2210           GTO RESTORE
2220 CREATIT   PUMD R10,+R6           ! SAVE OUR BASE
2230           ANM R17,=77            ! CLEAN UP THE ERROR FLAG
2240           CLB R20                ! AND THE OTHER ONE
2250           STBD R20,=ERRORS       ! OUT IN RAM
2260           LDM R36,=COUNT         ! GET THE REL ADDRESS OF ROUTINE
2270           ADM R36,R10            ! MAKE IT ABSOLUTE
2280           STM R36,R45            ! SET IT
2290           LDB R47,=236           ! LOAD A RTN OPCODE
2300           LDB R44,=316           ! LOAD A JSB OPCODE
2310           STMD R44,=IOTRFC       ! TAKE THE HOOK
2320           LDMD R72,=LLDCOM       ! SAVE LIST POINTERS
2330           PUMD R72,+R6           !   ON THE RTN STACK
2340           CLM R70                ! ZERO THE SELECT CODE
2350           STMD R70,=SCTEMP       ! SET THE SELECT CODE
2360           STMD R75,=NXTDAT       ! INITIALIZE BYTE COUNT TO 0
2370           PUBD R16,+R6           ! SAVE CSTAT
2380           LDB R16,=2             ! FAKE RUN MODE
2390           JSB =LSSET             ! LIST THE PROGRAM
2400           POBD R16,-R6           ! RESTORE CSTAT
2410           POMD R72,-R6           ! RESTORE THE LIST POINTERS
2420           STMD R72,=LLDCOM       ! RESET FIRST/LAST LINE POINTERS
2430           CLB R50                ! FOR THE MULTI-BYTE ADDS
2440           LDMD R45,=NXTDAT       ! GET THE BYTE COUNT
2450           JZR CALCRTN            ! JIF NOTHING THERE
2460           ADM R45,R46            ! WE NEED TO ADD THREE BYTES FOR EACH
2470           ADM R45,R46            !   RECORD BECAUSE OF THE HEADER USED EACH
2480           ADM R45,R46            !   TIME A STRING CROSSES RECORD BOUNDARY
2490           ADM R45,=3,0,0         ! AN EXTRA THREE
2500           TSB R45                ! IS IT ZERO?
2510           JZR NOINC              ! JIF YES
2520           ICM R46                ! ELSE ROUND IT UP
2530 NOINC     LDM R55,R46            ! SET IT FOR CONBI3
2540           JSB =CONBI3            ! CONVERT IT TO FLOATING-POINT
2550           POMD R10,-R6           ! RECOVER OUR BASE
2560           PUMD R10,+R6           ! SAVE IT AGAIN
2570           LDMD R53,X10,FILENAME  ! GET THE FILE NAME
2580           PUMD R53,+R12          ! PUSH IT TO STACK
2590           PUMD R40,+R12          ! PUSH THE NUMBER OF RECORDS DESIRED
2600           LDM R54,=377,56C,2C,0  ! MAKE 256 BYTE RECORDS
2610           PUMD R50,+R12          ! PUSH IT TO THE STACK
2620           JSB =ROMJSB            ! SELECT THE ROM
2630           DEF MSCRE.             ! CREATE THE FILE
2640           VAL MSROM#             ! ROM #
2650           POMD R10,-R6           ! RECOVER OUR BASE
2660           CMB R17,=300           ! ANY ERRORS ON THE CREATE?
```

```
2670          JCY SAVEX            ! JIF YES
2680          PUMD R10,+R6         ! SAVE OUR BASE
2690          JSB X10,ASNPRT       ! ASSIGN THE BUFFER AND DO THE MSPRNT
2700          POMD R10,-R6         ! RECOVER OUR BASE
2710          CMB R17,=300         ! ANY ERRORS IN THE ASSIGN?
2720          JNC PRINT            ! JIF NO
2730 SAVEX    GTO RESTORE          ! BAIL OUT
2740 ASNPRT   STMD R12,=TOS        ! MAKE SURE STACK LOOKS GOOD
2750          CLM R50              ! FIX UP FOR REAL 1
2760          LDB R57,=10C
2770          PUMD R50,+R12        ! PUSH IT TO THE STACK
2780          LDMD R53,X10,FILENAME ! GET THE FILE NAME
2790          PUMD R53,+R12        ! PUSH IT TO THE STACK
2800          JSB =ROMJSB          ! SELECT THE ROM
2810          DEF ASSIG.           ! ASSIGN THE BUFFER
2820          VAL MSROM#           ! ROM#
2830          CMB R17,=300         ! ANY ERRORS?
2840          JCY ASNRTN           ! JIF YES, DO NO MORE
2850          CLM R40              ! ELSE MAKE A 1
2860          LDB R47,=10C         !  (FLOATING POINT 1)
2870          PUMD R40,+R12        ! PUSH IT TO THE STACK
2880          JSB =ROMJSB          ! SELECT THE ROM
2890          DEF MSPRNT           ! DO THE READ#
2900          VAL MSROM#           ! ROM #
2910 ASNRTN   RTN                  ! DONE
2920 PRINT    LDM R36,=SAVERECS    ! GET THE REL ADDRESS OF OUR ROUTINE
2930          ADM R36,R10          ! MAKE IT ABSOLUTE
2940          STM R36,R45          ! SET IT
2950          LDB R47,=236         ! LOAD A RTN OPCODE
2960          LDB R44,=316         ! LOAD A JSB OPCODE
2970          STMD R44,=IOTRFC     ! TAKE THE HOOK
2980          PUBD R16,+R6         ! SAVE CSTAT
2990          LDB R16,=2           ! FAKE RUN MODE
3000          PUMD R10,+R6         ! SAVE OUR BASE
3010          JSB =LSSET           ! LIST AND PRINT# IT
3020          CLM R36              ! LINE LEN OF 0
3030          POMD R10,-R6         ! RECOVER OUR BASE
3040          PUMD R10,+R6         ! SAVE IT AGAIN
3050          JSB X10,SAVERECS     ! PRINT A NULL STRING AT THE END
3060          JSB =ROMJSB          ! SELECT THE ROM
3070          DEF PREOL.           ! DO THE END OF LINE PRINTING
3080          VAL MSROM#           ! ROM #
3090          POMD R10,-R6         ! RECOVER OUR BASE
3100          POBD R16,-R6         ! RESTORE CSTAT
3110          JSB X10,CLOSE        ! CLOSE THE FILE
3120 RESTORE  LDMD R71,X10,SAVIOTFC ! GET THE OLD HOOK
3130          STMD R71,=IOTRFC     ! RESTORE IT
3140          LDMD R70,X10,SAVSCTEM ! GET THE OLD SELECT CODE
3150          STMD R70,=SCTEMP     ! RESTORE IT
3160 FINMSG   JSB =CLEAR.          ! CLEAR THE CRT
3170          LDM R26,=MESAGE      ! LOAD THE ADDRESS OF THE MSG
3180          ADM R26,R10          ! MAKE IT ABSOLUTE
3190          LDM R36,=4,0         ! LOAD THE LEN
3200          JSB =OUTSTR          ! OUTPUT THE STRING
3210          RTN                  ! DONE
3220 MESAGE   ASC "DONE"
3230 !************************************************************************
3240 CLOSE    CLM R40              ! NEED ANOTHER 1
```

```
3250          LDB R47,=10C             ! FINISH THE 1
3260          PUMD R40,+R12            ! PUSH TO STACK
3270          LDM R46,=1,0             ! LENGTH OF THE "*" STRING
3280          PUMD R46,+R12            ! PUSH IT TO STACK
3290          LDM R45,=STAR            ! ADDRESS OF THE ASTERISK
3300          BYT 0                    ! NEED A THREE BYTE ADDRESS
3310          ADM R45,R10              ! MAKE IT ABSOLUTE
3320          CLB R47                  ! CLEAN UP THE MS BYTE
3330          PUMD R45,+R12            ! PUSH THE ADDRESS
3340          PUMD R10,+R6             ! SAVE OUR BASE
3350          JSB =ROMJSB              ! SELECT THE ROM
3360          DEF ASSIG.               ! CLOSE THE BUFFER
3370          VAL MSROM#               ! ROM# TO SELECT
3380          POMD R10,-R6             ! RECOVER THE BASE
3390          RTN                      ! DONE
3400 !*********************************************************************
3410 COUNT    BIN                      ! FOR THE MATH
3420          CLB R40                  ! FOR THE MULTI-BYTE ADD
3430          ADM R36,=4,0             ! ADD SOME FOR THE HEADER
3440          LDMD R45,=NXTDAT         ! GET THE PREVIOUS COUNT
3450          ADM R45,R36              ! ADD THE CURRENT LINE LEN
3460          STMD R45,=NXTDAT         ! SAVE THE NEW COUNT
3470          RTN                      ! DONE
3480 !*********************************************************************
3490 SAVERECS PUMD R36,+R12            ! PUSH THE LEN OF THE LINE
3500          STM R26,R24              ! COPY OF START
3510          ADM R26,R36              ! MOVE TO END OF STRING
3520          STM R26,R45              ! GET THE ADDRESS
3530          CLB R47                  ! CLEAR THE MOST SIGNIFICANT BYTE
3540          PUMD R45,+R12            ! PUSH THE ADDRESS
3550 SAVLOOP  CMM R24,R26              ! DONE?
3560          JCY PRINT-IT             ! JIF YES
3570          POBD R30,-R26            ! FETCH LAST BYTE
3580          LDBD R31,R24             ! FETCH FIRST BYTE
3590          STBD R31,R26             ! SWAP THEM
3600          PUBD R30,+R24            ! DITTO
3610          JMP SAVLOOP              ! LOOP TIL DONE
3620 PRINT-IT JSB =ROMJSB              ! SELECT THE ROM
3630          DEF PRSTR.               ! PRINT THE STRING
3640          VAL MSROM#               ! ROM#
3650          RTN
3660 !*********************************************************************
3670          BYT 141                  ! BASIC STATEMENT, LEGAL AFTER THEN
3680 GET.     BIN                      ! FOR ADDRESS MATH
3690          LDMD R10,=BINTAB         ! LET'S GET OUR BASE
3700          POMD R43,-R12            ! GET THE FILE NAME
3710          STMD R43,X10,FILENAME    ! SAVE IT AWAY
3720          CLB R16
3730          JSB =FXLEN.              ! MAKE SURE THE PROGRAM'S DEALLOCATED
3740          JSB =CLEAR.              ! CLEAR THE SCREEN
3750          LDM R26,=GETTING         ! GET ADDRESS OF MESSAGE
3760          ADMD R26,=BINTAB         ! MAKE IT ABSOLUTE
3770          LDM R36,=17,0            ! LOAD THE LENGTH OF THE MESSAGE
3780          JSB =OUTSTR              ! OUTPUT THE MESSAGE
3790          JSB =DECUR2              ! GET RID OF THE CURSOR
3800          JSB =DNCURS              ! MOVE DOWN ONE LINE
3810 BIN1     LDM R10,R4               ! GET THE PC
3820          BIN                      ! GOOD FOR ADDRESS MATH
```

```
4420          LDMD R24,R26          ! GET THE LEN OF THE STRING READ
4430          JZR GOTBUF            ! JIF NO CHARACTERS
4440 SWAP     POBD R32,-R26         ! GET THE NEXT CHARACTER
4450          PUBD R32,+R30         ! PUSH IT TO INPUT BUFFER
4460          DCM R24               ! DECREMENT LEN COUNT
4470          JNZ SWAP              ! JIF MORE TO DO
4480 GOTBUF   LDM R36,R30           ! COPY END OF BUFFER PTR
4490          SBM R36,=INPBUF       ! MINUS THE START OF BUFFER
4500          STMD R36,X10,BUFLEN   ! SAVE IN CASE OF ERROR FOR PRINT
4510          LDB R24,=15           ! LOAD A CR CHARACTER
4520          PUBD R24,+R30         ! PUSH IT OUT FOR PARSER
4530          PUBD R25,+R6          ! SAVE A 0 FLAG ON R6 FOR ERROR TRAP
4540          CMB R36,=81D          ! DO WE NEED TO MOVE THE CURSOR DOWN?
4550          JNC PARSIT            ! JIF NO
4560          JSB =DNCURS           ! MOVE CURSOR DOWN A ROW
4570 PARSIT   CLB R16               ! FOR LINEDR
4580          LDMD R20,=ASNTBL      ! SAVE ASSIGN BUFFER POINTER
4590          PUMD R20,+R6          ! ON THE R6 STACK
4600          LDMD R42,=LAVAIL      ! SAVE SOME SYSTEM POINTERS
4610          PUMD R42,+R6          ! ON THE R6 STACK
4620          LDMD R42,=RTNSTK      ! SAVE SOME MORE
4630          PUMD R42,+R6          ! SAME PLACE
4640          LDMD R45,=LWAMEM      ! SAVE SOME MORE
4650          PUMD R45,+R6          ! AGAIN
4660          LDMD R45,=LAVAIL      ! MOVE LWAMEM
4670          STMD R45,=LWAMEM      !   UP TO LAVAIL
4680          JSB =RSETGO           ! RESET EVERYTHING UP
4690          JSB =PARSER           ! TRY TO PARSE THE LINE
4700          POMD R45,-R6          ! START RECOVERING THINGS
4710          STMD R45,=LWAMEM      !
4720          POMD R42,-R6          !
4730          STMD R42,=RTNSTK      !
4740          POMD R42,-R6          !
4750          STMD R42,=LAVAIL      !
4760          POMD R20,-R6          !
4770          STMD R20,=ASNTBL      !
4780          LDB R16,=1            !
4790          CMB R17,=300          ! ANY ERRORS?
4800          JCY FIXIT             ! JIF YES
4810          BIN                   ! CONFIRM MATH MODE
4820          DCM R6                ! THROW AWAY ERROR TRAP FLAG
4830          GTO OKGET             ! LOOP
4840 FIXIT    POBD R36,-R6          ! RECOVER ERROR TRAP FLAG
4850          JNZ ERREXIT           ! JIF TWO ERRORS
4860          ICB R36               ! SET FLAG
4870          PUBD R36,+R6          ! PUT IT BACK
4880          ANM R17,=77           ! CLEAR ERROR FLAGS
4890          CLM R40               ! CLEAR ERROR FLAGS
4900          STMD R40,=ERLIN#      ! CLEAR ERROR FLAGS
4910          STBD R40,=ERRTYP      ! CLEAR ERROR FLAGS
4920          LDM R24,=INPBUF       ! GET ADDRESS OF BUFFER
4930          STM R24,R22           ! COPY
4940          ICM R24               ! MOVE AHEAD TO THE FIRST CHARACTER
4950 MOVE-1   POBD R20,+R24         ! GET THE FIRST CHARACTER
4960          PUBD R20,+R22         ! MOVE IT BACK ONE PLACE
4970          CMB R20,=40           ! A BLANK ?
4980          JZR MOVE-1            ! JIF YES
4990          JSB =DIGIT            ! IS IT A DIGIT?
```

```
        SBM R10,=BIN1          ! GET OUR BASE ADDRESS
        STMD R10,=BINTAB       ! RESTORE BINTAB CASE 'FXLEN' DESTROYED
        JSB X10,ASNPRT         ! TRY TO OPEN THE FILE
        CMB R17,=300           ! ANY ERRORS?
        JNC OKGET              ! JIF NO, IT'S THERE
        LDMD R10,=BINTAB       ! GET OUR BASE
        GTO FINMSG             ! OUTPUT THE MESSAGE
OKGET   LDM R10,R4             ! GET PC
        BIN                    !
        SBM R10,=OKGET         ! GET OUR BASE ADDRESS
        STMD R10,=BINTAB       ! SET IT IN CASE PARSING BLEW IT AWAY
        LDMD R45,=NXTMEM       ! GET HIGH ADDRESS OF AVAILABLE SPACE
        SBMD R45,=LAVAIL       ! GET AVAILABLE MEMORY COUNT
        CMM R45,=0,2,0         ! ENOUGH MEMORY LEFT?
        JCY OKGET2             ! JIF YES
        JSB =ERROR             ! ELSE REPORT ERROR
        BYT 19D                ! MEM OVF
        GTO FINMSG             ! OUTPUT 'DONE' MESSAGE
GETDON  LDBD R40,=ERRORS       ! GET REASON FOR ERROR
        CMB R40,=107           ! END OF FILE ERROR?
        JZR EOFERR             ! JIF YES
        CMB R40,=110           ! END OF RECORD ERROR?
        JNZ BADERR             ! JIF NO, LET IT GO
EOFERR  CLM R40                ! ELSE CLEAR ERROR FLAGS
        STMD R40,=ERLIN#       !  ---
        STBD R40,=ERRTYP       ! ---
        ANM R17,=77            !    AND IN XCOM
BADERR  JSB =ST240+            ! SET IMMEDIATE BREAK BITS
BIN5    LDM R10,R4             ! COPY OF PC
        BIN                    ! FOR ADDRESS MATH
        SBM R10,=BIN5          ! GET BASE ADDRESS
        JSB X10,CLOSE          ! CLOSE THE FILE
        GTO FINMSG             ! OUTPUT THE 'DONE' MESSAGE
OKGET2  LDMD R12,=TOS          ! RESET STACK POINTER
        LDM R45,=BUFFER        ! GET THE ADDRESS OF THE BUFFER
        BYT 0                  ! AS A THREE BYTE QUANTITY
        ADMD R45,=BINTAB       ! MAKE IT ABSOLUTE
        PUMD R45,+R12          ! PUSH TO STACK
        LDM R51,=240,0,0,0,0,0,200 ! TOTAL SIZE, NAME PTR, HEADER
        PUMD R51,-R45          ! FAKE VARIABLE HEADER AREA
        LDM R64,=0,0,240,0     ! CURRENT LEN, MAX LEN
        PUMD R64,-R45          ! MORE VARIABLE HEADER STUFF
        PUBD R57,+R12          ! PUSH STUFF FOR STOST: HEADER
        PUMD R66,+R12          !          MAX LEN STRING VAR (0,1)
        PUMD R45,+R12          !          ADDRESS OF FIRST BYTE OF $ VAR
        PUMD R66,+R12          !          MAX LEN TO STORE INTO
        PUMD R45,+R12          !          ADDRESS TO STORE INTO
        STMD R45,X10,BUFADR    ! SAVE BUFFER ADDRESS
        JSB =ROMJSB            ! CALL A BANK SELECT ROM
        DEF RDSTR.             ! READ A STRING FROM THE FILE
        VAL MSROM#             ! IT'S THE MASS STORAGE ROM
        CMB R17,=300           ! ANY ERRORS ?
        JCY GETDON             ! JIF YES
        LDMD R10,=BINTAB       ! ELSE GET BASE ADDRESS
        LDMD R26,X10,BUFADR    ! GET ADDRESS OF BUFFER
        BIN
        LDM R30,=INPBUF        ! GET ADDRESS OF INPUT BUFFER
        LDB R32,=40            ! LOAD A BLANK
        PUBD R32,+R30          ! PUSH IT TO BUFFER
```

```
5000          JEN MOVE-1              ! JIF YES
5010          LDB R20,=41             ! ELSE LOAD A !
5020          PUBD R20,-R22           ! PUSH IT TO THE HOLE
5030          JSB =PRINT.             ! SET THE SCTEMP SELECT CODE
5040 BIN3     LDM R10,R4              ! GET PC
5050          BIN                     ! CALCULATE BASE IN CASE PARSER DESTROYED
5060          SBM R10,=BIN3           !     BINTAB
5070          LDMD R36,X10,BUFLEN     ! GET LENGTH OF BUFFER
5080          LDM R26,=INPBUF         ! GET THE START ADDRESS
5090          JSB =DRV12.             ! PRINT THE LINE
5100          GTO PARSIT              ! GOT PARSE IT AS A COMMENT
5110 ERREXIT  LDM R10,R4              ! GET CURRENT ADDRESS
5120          BIN                     ! FOR ADDRESS MATH
5130          SBM R10,=ERREXIT        ! GET BPGM'S BASE ADDRESS
5140          GTO FINMSG              ! GO DISPLAY 'DONE' MESSAGE
5150 !**********************************************************************
5160          BYT 0,56
5170 REVISON. BIN                     ! FOR ADDRESS MATH
5180          LDM R43,=40D,0          ! LEN OF STRING
5190          DEF DATE                ! ADDRESS AS TWO BYTE REL
5200          BYT 0                   ! THERE'S THE THIRD BYTE
5210          ADMD R45,=BINTAB        ! NOW IT'S ABSOLUTE
5220          PUMD R43,+R12           ! PUSH TO RETURN STACK
5230          RTN                     ! DONE
5240          ASC "81.202 .veR 2891 .oC drakcaP-ttelweH )c("
5250 DATE     BSZ 0
5260 !**********************************************************************
5270 SAVING   ASC "SAVE IN PROGRESS"
5280 GETTING  ASC "GET IN PROGRESS"
5290 DONE     ASC "DONE"
5300          ASC "*"
5310 STAR     BSZ 0
5320 SAVIOTFC BSZ 7
5330 SAVSCTEM BSZ 10
5340 FILENAME BSZ 5
5350 BUFADR   BSZ 3
5360 BUFLEN   BSZ 2
5370          BSZ 300
5380 BUFFER   BSZ 0
5390 !**********************************************************************
5400 ASNTBL   DAD 100125
5410 ASSIG.   DAD 65466
5420 BINTAB   DAD 104070
5430 CALVRB   DAD 100030
5440 CLEAR.   DAD 14225
5450 CONBI3   DAD 4516
5460 DECUR2   DAD 13467
5470 DIGIT    DAD 21710
5480 DNCURS   DAD 13751
5490 DRV12.   DAD 6722
5500 ERLIN#   DAD 100114
5510 ERROR    DAD 10223
5520 ERROR+   DAD 10220
5530 ERRORS   DAD 100123
5540 ERRTYP   DAD 100124
5550 FLDCOM   DAD 100053
5560 FXLEN    DAD 31001
5570 GO12N    DAD 24707
```

```
5580 INPBUF    DAD 100236
5590 IOTRFC    DAD 103643
5600 LAVAIL    DAD 100025
5610 LLDCOM    DAD 100050
5620 LSSET     DAD 6445
5630 LWAMEM    DAD 100041
5640 MSCRE.    DAD 65176
5650 MSPRNT    DAD 66221
5660 MSPUR.    DAD 64604
5670 MSROM#    DAD 320
5680 NXTDAT    DAD 101645
5690 NXTMEM    DAD 100022
5700 ONEI      DAD 56736
5710 OUTSTR    DAD 14020
5720 PARSER    DAD 20000
5730 PREOL.    DAD 70464
5740 PRINT.    DAD 71332
5750 PRSTR.    DAD 66662
5760 PTR2-     DAD 177715
5770 PTR2+     DAD 177716
5780 RDSTR.    DAD 67314
5790 ROMJSB    DAD 6223
5800 RSETGO    DAD 5700
5810 RTNSTK    DAD 100033
5820 SCTEMP    DAD 101721
5830 ST240+    DAD 21067
5840 STREX+    DAD 23721
5850 TOS       DAD 101744
5860 FIN       FIN
```

REFERENCE MATERIAL

8.1 Overview

This section consists of:

- An alphabetical listing of the global file.
- System operation and routines.
- Parsing flow diagrams.
- General hook flowcharts for the following:

  CHIDLE
  DCIDLE
  IOSP
  IOTRFC
  IRQ2Ø
  KYIDLE
  PRSIDL
  RMIDLE
  SPARØ and SPAR1

- System run time table tokens and attributes.
- Error messages.
- The assembler instruction set.
- An assembler instruction coding table.
- A keycode table.
- Some programming hints.

8.2 The Global File

The global file as it appears on the disc is listed here. It gives the permanent addresses in memory of many of the system routines. The global file also contains locations of system pointers, buffers, variables, and constants which may be referenced in a binary 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 on a disc, 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.

```
     Name       Address                    Description

1000 !***********************************************************************
1010 !*                                                                     *
1020 !*  HP-87 GLOBAL FILE FOR USE WITH THE ASSEMBLER ROM.                  *
1030 !*                                                                     *
1040 !*  (c) 1982 Hewlett-Packard Co.                                       *
1050 !*                                                                     *
1060 !*                                                                     *
1070 !***********************************************************************
1080 !*NOTE: Beware of looking up a routine in the global file and using    *
1090 !*it without also looking up the documentation. This is especially    *
1100 !*true if the routine has an entry point address between 60000 and    *
1110 !*77777, as it may need to be called through ROMJSB.                   *
1120 !***********************************************************************
1130         GLO
1140 ABS5    DAD 54525              ! ABS FUNCTION RUNTIME CODE
1150 ACTB-3  DAD 177515             ! I/O MODULE ADDRESSES
1160 ACTB-6  DAD 177512             ! I/O MODULE ADDRESSES
1170 ACTBAS  DAD 177520             ! I/O MODULE ADDRESSES
1180 ACTBS+  DAD 177521             ! I/O MODULE ADDRESSES
1190 ACTMSU  DAD 103560             ! ACTIVE MSUS FOR MASS STORAGE ROM
1200 ADD10   DAD 53030              ! ADD TWO REAL NUMBERS IN R40 AND R50
1210 ADDROI  DAD 52745              ! ADD 2 REAL OR INTEGER NUMBERS OFF STACK
1220 AGLBAS  DAD 103416             ! PLOTTER ROM STOLEN RAM BASE ADDRESS
1230 ALFA    DAD 21656              ! CHECK TO SEE IF R20 IS ASCII A-Z OR a-z
1240 ALFAL.  DAD 12466              ! FORCE ALPHA ALL MODE
1250 ALPHA   DAD 12542              ! FORCE ALPHA OR ALPHA ALL IF NOT GRAPHALL
1260 ALPHA.  DAD 12413              ! FORCE ALPHA NORMAL
1270 APRBAS  DAD 103420             ! ADVANCED PROG ROM STOLEN RAM BASE ADDR.
1280 ASIZE   DAD 104744             ! # OF BYTES IN ALPHA (4K OR 16K)
1290 ASMBAS  DAD 103426             ! ASSEMBLER ROM
1300 ASNTBL  DAD 100125             ! 24 BYTES ASSIGN FILES
1310 ASSIG.  DAD 65466              ! ASSIGN A DISC BUFFER TO A FILE
1320 ATN2.   DAD 77157              ! ATN2 FUNCTION
1330 AUTO#   DAD 100103             ! AUTO LINE # LAST VAL
```

Name	Address	Description
1340 AUTOI	DAD 100106	! AUTO LINE # INC
1350 BEEP.	DAD 10361	! BEEP STATEMENT
1360 BINBAS	DAD 104073	! 5 BP'S ADDRESSES
1370 BINTAB	DAD 104070	! 3 BYTES BP BASE ADDRS.
1380 BKSPC	DAD 11520	! BACKSPACE KEY RUNTIME
1390 BLKLIN	DAD 14165	! BLANK LINE ON CRT
1400 BOS	DAD 105350	! FIXED SIZE R12 STACK
1410 BOVAR	DAD 100014	! BEGIN OF LOCAL VAR
1420 BPINI	DAD 6113	! CALL INIT ROUTINES IN BINARY PROGRAMS
1430 BSRBAS	DAD 103422	! PROGRAM DEVELOPEMENT ROM STOLEN RAM BASE
1440 BYTCRT	DAD 14004	! SEND ADDRESS TO CRTBAD AND CRTBYT
1450 CALVRB	DAD 100030	! START OF CALC VARIABLES
1460 CEIL10	DAD 54412	! CEIL FUNCTION RUNTIME CODE
1470 CHIDLE	DAD 103670	! CHAR. EDITOR INTERCEPT RAM HOOK
1480 CHKSTS	DAD 13204	! WAIT FOR CRT CONTROLLER NOT BUSY
1490 CHSRUI	DAD 52672	! CHANGE SIGN OF REAL OR INTEGER NUMBER
1500 CLEAR.	DAD 14225	! CLEAR ALPHA DISPLAY
1510 CLKDAT	DAD 177413	! CLOCK DATA
1520 CLKSTS	DAD 177412	! CLOCK STATUS
1530 CLREOL	DAD 13447	! CLEAR TO END OF LINE ON CRT
1540 CNTRTR	DAD 13245	! COUNT RETRACES (60 / SECOND)
1550 COLUMN	DAD 14206	! FIND WHAT COLUMN ON ALPHA DISPLAY
1560 COMMA$	DAD 72146	! PRINT STRING,
1570 COMMA.	DAD 72265	! PRINT NUMBER,
1580 CONBI3	DAD 4516	! CONVERT 3-BYTE BINARY # TO REAL
1590 CONBIN	DAD 4401	! CONVERT 2-BYTE BINARY # TO REAL
1600 CONCA.	DAD 76366	! CONCATENATE TWO STRINGS
1610 CONINT	DAD 45116	! CONVERT A REAL # TO A 15-BIT SIGNED BIN.
1620 CONTR.	DAD 61620	! I/O MODULE 'CONTROL' STATEMENT
1630 COS10	DAD 54353	! COSINE FUNCTION
1640 COT10	DAD 54333	! COTANGENT FUNCTION
1650 COUNTK	DAD 14411	! KEY REPEAT ROUTINE
1660 CPRBAS	DAD 103432	! CAPR ROM
1670 CRT.	DAD 57307	! 'CRT IS' STATEMENT
1680 CRTBAD	DAD 177701	! CRT BYTE ADDRESS
1690 CRTBLK	DAD 12246	! FILL ALPHA MEMORY WITH CHR$(13)'s
1700 CRTBYT	DAD 100206	! CRT BYTE ADDRESS
1710 CRTDAT	DAD 177703	! CRT DATA
1720 CRTINT	DAD 12176	! INITIALIZE CRT MEMORY
1730 CRTLST	DAD 101101	! # LINES ON CRT PAGE -1
1740 CRTPOF	DAD 12334	! POWER DOWN CRT HIGH VOLTAGE
1750 CRTPUP	DAD 12341	! POWER UP CRT HIGH VOLTAGE
1760 CRTRAM	DAD 100210	! CRT START ADDRESS (COPY IN RAM)
1770 CRTSAD	DAD 177700	! CRT START ADDRESS I/O ADDRESS
1780 CRTSTS	DAD 177702	! CRT STATUS I/O ADDRESS
1790 CRTUNW	DAD 12360	! UNBLANK THE CRT
1800 CRTWPO	DAD 12374	! BLANK THE CRT
1810 CRTWRS	DAD 101655	! CRT STATUS IN RAM
1820 CS.C.	DAD 100212	! CRT SELECT CODE (8 BYTES)
1830 CSEC10	DAD 54300	! COSECANT FUNCTION

	Name	Address	Description
1840	CSIZE.	DAD 66570	! 'CSIZE' STATEMENT
1850	CURS	DAD 14030	! TURN CURSOR ON
1860	CURSON	DAD 105347	! CURSOR ON FLAG
1870	CVNUM	DAD 72401	! FORMAT A REAL NUMBER FOR OUTPUT
1880	DALLED	DAD 101104	! DEALLOCATED FLAG
1890	DALLOC	DAD 47123	! DE-ALLOCATE THE BASIC PROGRAM
1900	DATE	DAD 101133	! JULIAN DAY YEAR
1910	DATE.	DAD 32073	! DATE FUNCTION
1920	DCIDLE	DAD 104035	! DCOMPILE HOOK
1930	DCLIN#	DAD 34607	! DECOMPILE A BASIC PROGRAM LINE NUMBER
1940	DCSLOP	DAD 35132	! REVERSE A STRING FROM EXTENDED MEMORY
1950	DECUR2	DAD 13467	! TURN CURSOR OFF
1960	DEFA+.	DAD 61576	! TURN MATH DEFAULTS ON
1970	DEFA-.	DAD 61604	! TURN MATH DEFAULTS OFF
1980	DEFAUL	DAD 100152	! DEFAULT ERROR FLAG
1990	DEFMSU	DAD 103477	! DEFAULT MSUS
2000	DEG.	DAD 62257	! PUTS THE COMPUTER IN DEGREES TRIG MODE
2010	DEG10	DAD 54736	! RADIANS TO DEGREES CONVERSION
2020	DFLAG	DAD 104224	! DIRECTION FLAG FOR DISC READ/WRITE
2030	DGHOOK	DAD 104044	! DIGITIZE HOOK FOR CRT DIGITIZING
2040	DIGIT	DAD 21710	! SEE IF R20 CONTAINS A DIGIT (ASCII CODE)
2050	DISBUF	DAD 100542	! DISPLAY BUFFER
2060	DISP.	DAD 71311	! SET SELECT CODE TO CRT IS DEVICE
2070	DISPLN	DAD 101136	! 1 BYTE DISPLAY LINE LENGTH
2080	DISPTR	DAD 100060	! DISP BUFFER PTR
2090	DIV10	DAD 52441	! DIVIDE 2 REAL NUMBERS IN R40 AND R50
2100	DIV2	DAD 52436	! DIVIDE 2 REAL OR INTEGER NUMBERS ON STAK
2110	DMNDCR	DAD 25175	! DEMAND CARRIAGE RTN, BANG (!), OR @ SIGN
2120	DNCUR.	DAD 13607	! MOVE CURSOR DOWN ON CURRENT CRT PAGE
2130	DNCURS	DAD 13751	! MOVE CURSOR DOWN IN ALPHA MEMORY
2140	DRAW.	DAD 64727	! DRAW A LINE ON THE CRT
2150	DRG	DAD 100160	! DEG/RAD/GRAD FLAG
2160	DRV12.	DAD 6722	! OUTPUT VECTOR ROUTINE
2170	EDMOD2	DAD 100122	! EDITOR MODE (INSERT/REPLACE)
2180	EMOVDN	DAD 32161	! EXTENDED MEMORY MOVDN
2190	EMOVUP	DAD 32231	! EXTENDED MEMORY MOVUP
2200	ENDSR	DAD 14750	! END OF SERVICE ROUTINE(FIX UP EMC'S DRP)
2210	EOJ2	DAD 14525	! END OF JOB (TURN OFF KEY)
2220	EOVAR	DAD 100017	! END OF LOCAL VARIABLE POINTER
2230	EPS10	DAD 54722	! EPS FUNCTION
2240	EQ$.	DAD 3564	! COMPARE TWO STRINGS FOR EQUAL
2250	EQ.	DAD 62623	! COMPARE TWO NUMBERS FOR EQUAL
2260	ERBEND	DAD 100542	! END ERROR BUFFER + 1
2270	ERLIN#	DAD 100114	! LINE# OF BAD LINE
2280	ERNUM#	DAD 100117	! ERROR NUMBER
2290	ERRBP#	DAD 103371	! BPGM # THAT REPORTS THE ERR
2300	ERRBUF	DAD 100476	! ERROR BUFFER (44 BYTES)
2310	ERROM#	DAD 100121	! ROM# OF LAST ERROR
2320	ERROR	DAD 10224	! REPORT ERROR ROUTINE
2330	ERROR+	DAD 10220	! REPORT ERROR AND THROW AWAY 1 RTN ADDR.
2340	ERRORS	DAD 100123	! RUN TIME ERRORS
2350	ERROM	DAD 100120	! ROM# OF ERROR

Name	Address	Description
ERRSC	DAD 101141	! ERROR SELECT CODE
ERRTYP	DAD 100124	! ERROR TYPE
ERTEMP	DAD 104200	! 12 BYTES TEMP
EXEC	DAD 72	! BEGINNING OF MAIN EXEC LOOP
EXP5	DAD 53174	! EXP FUNCTION (e^X)
EXSTAT	DAD 177426	! EXTENDED IO STATUS
EXTFIL	DAD 110010	! EXTENDED FILE TYPE TABLE
FASTBS	DAD 11565	! FAST BACKSPACE (SHIFTED BACKSPACE KEY)
FBPGM	DAD 50333	! FIND BINARY PROGRAM (BY BPGM #)
FETAVA	DAD 45505	! FETCH ARRAY VARIABLE ADDRESS
FETSVA	DAD 45305	! FETCH SIMPLE VARIABLE ADDRESS
FILTYP	DAD 101671	! 1 BYTE TAPE, TEMP
FLDCOM	DAD 100053	! FIRST LINE DECOMPILE
FLIP	DAD 14544	! TOGGLE THE KEYBOARD 'FLIP' STATUS
FNAM	DAD 103503	! FILE NAME 1ST HALF
FNAM+5	DAD 103510	! FILE NAME 2ND HALF
FNDLIN	DAD 32355	! FIND A BASIC PROGRAM LINE IN MEMORY
FORMAR	DAD 27034	! PARSE AN ARRAY REFERENCE
FP5	DAD 54665	! FRACTIONAL PART FUNCTION
FRAME.	DAD 66165	! FRAME THE CRT
FWBIN	DAD 100044	! FWA USER BIN PROG
FWCURR	DAD 100006	! PTR TO CURRENT PGM
FWPRGM	DAD 100003	! FWA PROGRAM AREA
FWROM	DAD 110130	! FWA USER PROGRAM ROMRAM
FWUSER	DAD 100000	! FWA USER AREA
G$N	DAD 24543	! GET STRING & NUMERIC
G$N+NN	DAD 24642	! GET STRING & NUMERIC WITH OPTIONALS
G/A	DAD 11606	! TOGGLE BETWEEN GRAPH AND ALPHA
G012N	DAD 24707	! GET 0, 1, OR 2 NUMERIC VALUES
G01N	DAD 24726	! GET 0 OR 1 NUMERIC VALUES
G0OR2N	DAD 24744	! GET 0 OR 2 NUMERIC VALUES
G12OR4	DAD 24772	! GET 1, 2, OR 4 NUMERIC VALUES
G1OR2N	DAD 24761	! GET 1 OR 2 NUMERIC VALUES
GCHAR	DAD 21636	! GET A CHARACTER AT PARSE TIME
GCLR.	DAD 62214	! CLEAR THE GRAPHICS CRT DISPLAY
GEMINI	DAD 104157	! GEMINI FLAG
GEQ$.	DAD 3667	! COMPARE FOR GREATER THAN OR EQUAL TO
GEQ.	DAD 62734	! COMPARE TWO NUMBERS FOR >=
GET)	DAD 23450	! GET A CLOSE PARENTHESIS
GET1N	DAD 24557	! PARSE ONE NUMBER
GET2N	DAD 24630	! PARSE TWO NUMBERS
GET4N	DAD 24635	! PARSE FOUR NUMBERIC PARAMETERS
GETCMA	DAD 23477	! DEMAND A COMMA AT PARSE TIME
GETPA?	DAD 24740	! GET SOME OPTIONAL PARAMETERS
GETPAR	DAD 24562	! GET A SET NUMBER OF NUMERIC PARAMETERS
GINTDS	DAD 177401	! GLOBAL INTERRUPT DISABLE
GINTEN	DAD 177400	! GLOBAL INTERRUPT ENABLE
GLINE	DAD 104740	! NUMBER OF DOTS ON A LINE OF GRAPH SCREEN
GLOAD	DAD 72510	! 'GLOAD' STATEMENT
GNAM	DAD 103515	! FOR MASS STORAGE COPY ,RENAME, ETC.
GNAM+5	DAD 103522	! FOR MASS STORAGE COPY, RENAME, ETC.
GOTOSU	DAD 30317	! PARSE A GOTO/GOSUB LINE NUMBER OR LABEL

Name	Address	Description
2880 GR$.	DAD 03614	! COMPARE STRINGS FOR GREATER THAN
2890 GR.	DAD 62705	! COMPARE NUMBERS FOR GREATER THAN
2900 GRAD.	DAD 62274	! SET THE COMPUTER TO GRAD MODE
2910 GRAFA.	DAD 12626	! FORCE GRAPH ALL MODE
2920 GRAPH	DAD 12560	! SWITCH TO GRAPH UNLESS IN ALPHA ALL
2930 GRAPH.	DAD 12574	! FORCE GRAPH NORMAL MODE
2940 GSIZE	DAD 104742	! # OF BYTES IN GRAPH SCREEN (12K OR 16K)
2950 GSTOR.	DAD 72711	! 'GSTORE' STATEMENT
2960 HLFLIN	DAD 14110	! DISP STRING WITHOUT CR AND LF
2970 HMCURS	DAD 13661	! HOME CURSOR ON CURRENT CRT PAGE
2980 HORN	DAD 10400	! LOWER LEVEL 'BEEP' ENTRY POINT
2990 ICOS	DAD 77254	! ARC COSINE FUNCTION
3000 IDRAW.	DAD 64706	! 'IDRAW' STATEMENT
3010 IMERR	DAD 103724	! IMAGE ERROR INTERCEPT RAM HOOK
3020 IMOVE.	DAD 64643	! 'IMOVE' STATEMENT
3030 INCHR	DAD 14262	! READ ONE CHARACTER IN FROM CRT MEMORY
3040 INF10	DAD 54321	! INFINITY FUNCTION (RETURNS BIGGEST #)
3050 INIT.	DAD 1241	! 'INIT' KEY EXECUTION
3060 INPB-3	DAD 100233	! 3 PERMANENT BYTES IN FRONT OF INPBUF
3070 INPBUF	DAD 100236	! PARSER INPUT BUFFER
3080 INPCOM	DAD 100167	! INPUT COMPLETION ADDRESS
3090 INPR10	DAD 101717	! R10 SAVE DURING INPUT
3100 INPTOS	DAD 100204	! INPUT TOP OF STAK
3110 INPTR	DAD 101143	! INPUT LINE POINTER
3120 INPUT.	DAD 16314	! INPUT RUNTIME ROUTINE
3130 INT5	DAD 54572	! INT FUNCTION
3140 INTDIV	DAD 54601	! INTEGER DIVISION (\) RUNTIME
3150 INTEGR	DAD 21331	! GET AN INTEGER AT PARSE TIME
3160 INTMUL	DAD 53673	! MULTIPLY TWO BINARY NUMBERS
3170 INTORL	DAD 57125	! CONVERT A TAGGED INTEGER TO A REAL
3180 INTRSC	DAD 177500	! I/O CARDS SELECT CODE ADDRESS
3190 IOBASE	DAD 103414	! I/O ROM BASE RAM POINTER
3200 IOBITS	DAD 101140	! 1 BYTE I/O
3210 IODATA	DAD 177422	! I/O DATA
3220 IOINTC	DAD 177421	! I/O CONTR-INTRUPT
3230 IOSP	DAD 103652	! I/O SERVICE POINTER RAM HOOK
3240 IOSTAT	DAD 177420	! I/O STATUS
3250 IOSW	DAD 100163	! I/O SERVICE WORD
3260 IOTRFC	DAD 103643	! 7 BYTES TRAFFIC INTERCEPT
3270 IP5	DAD 54770	! IP FUNCTION RUNTIME CODE
3280 IPLOT.	DAD 64660	! 'IPLOT' STATEMENT
3290 IRQ20	DAD 103742	! I/O INTERRUPT RAM HOOK
3300 IRQ20+	DAD 103751	! I/O INTERRUPT RAM HOOK
3310 IRQPAD	DAD 103757	! I/O INTERRUPT RAM HOOK
3320 IRQRTN	DAD 103760	! I/O INTERRUPT RAM HOOK
3330 ISIN	DAD 77244	! ARC SINE FUNCTION
3340 ITAN	DAD 77264	! ARC TANGENT FUNCTION
3350 KEYCNT	DAD 100153	! KEYBOARD REPEAT COUNTER
3360 KEYCOD	DAD 177403	! KEYBOARD CODE AND EOJOB I/O ADDRESS
3370 KEYHIT	DAD 101142	! KEYBOARD ASCII
3380 KEYLA.	DAD 13360	! KEY LABEL RUNTIME ROUTINE
3390 KEYSTS	DAD 177402	! KEYBOARD STATUS I/O ROUTINE

Name	Address	Description
3400 KEYTAB	DAD 102016	! BASE ADDR KEY TABL
3410 KRPET1	DAD 100154	! MAJOR KYBD REPEAT
3420 KRPET2	DAD 100155	! MINOR KYBD REPEAT
3430 KYIDLE	DAD 103677	! KEYBOARD INTERCEPT
3440 LABEL.	DAD 67262	! 'LABEL' STATEMENT
3450 LASTIN	DAD 100475	! END OF INPUT BUFFER
3460 LAVAIL	DAD 100025	! LAST AVAIL WD IN PGM AREA
3470 LDIR.	DAD 67052	! 'LDIR' STATEMENT
3480 LEGCA2	DAD 101525	! CALC KEYLABELS (BTM ROW)
3490 LEGCAL	DAD 101405	! CALC KEYLABELS (TOP ROW)
3500 LEGEN2	DAD 101265	! RUN KEYLABLES (BTM ROW)
3510 LEGEND	DAD 101145	! RUN KEYLABELS (TOP ROW)
3520 LEQ$.	DAD 3656	! COMPARE STRINGS FOR LESS THAN OR EQUAL
3530 LEQ.	DAD 62662	! COMPARE NUMBERS FOR LESS THAN OR EQUAL
3540 LINELN	DAD 101714	! DEVICE LINE LENGTH
3550 LINET.	DAD 66336	! 'LINE TYPE' STATEMENT
3560 LIST.	DAD 6352	! 'LIST' STATEMENT
3570 LLDCOM	DAD 100050	! LAST LINE DECOMPILE
3580 LLN-1	DAD 104231	! PGSIZE - ONE LINE
3590 LLN-2	DAD 104233	! PGSIZE - TWO LINES
3600 LN5	DAD 52346	! NATURAL LOGARITHM FUNCTION
3610 LNTYPE	DAD 104750	! LINE TYPE POINTER TABLE
3620 LOGT5	DAD 52515	! BASE 10 LOGARITHM FUNCTION
3630 LSTBUF	DAD 103200	! LWA + 1 DISC BUFFER
3640 LSTDAT	DAD 101650	! LAST DATA ADDR. FOR DISC READ/WRITE
3650 LT$.	DAD 3635	! COMPARE STRINGS FOR LESS THAN
3660 LT.	DAD 62643	! COMPARE NUMBERS FOR LESS THAN
3670 LTCUR.	DAD 13623	! LEFT CURSOR ON CURRENT PAGE
3680 LTCURS	DAD 13757	! LEFT CURSOR IN ALPHA MEMORY
3690 LTYPE#	DAD 104537	! LINE TYPE #
3700 LWAMEM	DAD 100041	! LAST WORD AVAILABLE USER MEMORY
3710 MAX10	DAD 56144	! MAX FUNCTION RUNTIME CODE
3720 MBASE	DAD 103424	! MATRIX ROM STOLEN RAM BASE ADDRESS
3730 MIN10	DAD 56125	! MIN FUNCTION RUNTIME CODE
3740 MLAD	DAD 177424	! SERIAL POLL REGISTER
3750 MOD10	DAD 52541	! MOD FUNCTION RUNTIME ADDRESS
3760 MODADR	DAD 13255	! KEEPING ADDRESS IN ALPHA MEMORY ON CRT
3770 MOVCRS	DAD 13771	! MOVE CURSOR BY SPECIFIED AMOUNT
3780 MOVDN	DAD 57172	! MOVE MEMORY CONTENTS WITH DECREASING PTR
3790 MOVE.	DAD 64634	! 'MOVE' STATEMENT
3800 MOVUP	DAD 57232	! MOVE MEMORY CONTENTS WITH INCREASING PTR
3810 MPY10	DAD 53357	! MULTIPLY TWO REAL #'S IN R40 AND R50
3820 MPYROI	DAD 53517	! MULTIPLY TWO REAL OR INTEGER #s ON STACK
3830 MSBASE	DAD 103412	! MASS STORAGE ROM STOLEN RAM BASE ADDRESS
3840 MSCRE.	DAD 65176	! CREATE RUNTIME CODE
3850 MSHIGH	DAD 103764	! MS HIGH LEVEL HOOK
3860 MSLOW	DAD 103773	! MS LOW LEVEL HOOK
3870 MSPRNT	DAD 66221	! PART OF PRINT# RUNTIME CODE
3880 MSPUR.	DAD 64604	! PURGE RUNTIME CODE
3890 MSREN.	DAD 64724	! 'RENAME' STATEMENT
3900 MSTIME	DAD 104002	! MS TIMEOUT HOOK
3910 NARRE+	DAD 23461	! SCAN AND PARSE A NUMERIC ARRAY REFERENCE

Name	Address	Description
3920 NARREF	DAD 23465	! PARSE A NUMERIC ARRAY REFERENCE
3930 NUMCON	DAD 23551	! PARSE A NUMERIC CONSTANT
3940 NUMVA+	DAD 22403	! SCAN AND PARSE A NUMERIC EXPRESSION
3950 NUMVAL	DAD 22406	! PARSE A NUMERIC EXPRESSION
3960 NXTDAT	DAD 101645	! NEXT DATA ADDRESS FOR DISK READ/WRITE
3970 NXTMEM	DAD 100022	! NEXT BYTE AVAILABLE MEMORY
3980 NXTRTN	DAD 100036	! NEXT AVAILABLE GOSUB/RTN
3990 ONEB	DAD 12153	! GET 1 NUMBER OFF STACK AS SIGNED BINARY
4000 ONEI	DAD 56736	! GET 1 NUMBER OFF STACK AS TAGGED INTEGER
4010 ONER	DAD 56777	! GET 1 NUMBER OFF STACK AS FLOATING POINT
4020 ONEROI	DAD 57035	! GET 1 NUMBER OFF STACK AS REAL OR INTEGR
4030 ONEX	DAD 56673	! GET 1 NUMBER OFF STACK AS UNSIGNED BIN.
4040 ONFLAG	DAD 100065	! ON GOSUB FLAG
4050 OPTBAS	DAD 100175	! 2 BYTE PERMANENT OPTION BASE
4060 OUTCH1	DAD 14130	! OUTPUT A BYTE TO THE CRT
4070 OUTCHR	DAD 14143	! OUTPUT A CHARACTER TO CRT
4080 OUTSTR	DAD 14020	! OUTPUT A STRING TO CRT
4090 P.BUFF	DAD 101706	! INDIRECT BUFFER POINTERS
4100 P.FLAG	DAD 101712	! INDIRECT BUFFER FLAG
4110 P.PTR	DAD 101710	! INDIRECT BUFFER POINTER
4120 P.TYPE	EQU 6	! OFFSET INTO BASIC PCB TO GET TYPE BYTE
4130 PAGES.	DAD 12756	! PAGESIZE RUNTIME CODE
4140 PAGES1	DAD 13001	! PAGESIZE 16
4150 PAGES2	DAD 13103	! PAGESIZE 24
4160 PARSER	DAD 20000	! SYSTEM PARSER
4170 PEN#	DAD 104535	! (PEN #) * 3 FOR INDEXING
4180 PGSIZE	DAD 104227	! # OF BYTES / PAGE
4190 PI10	DAD 54374	! PI FUNCTION RUNTIME CODE
4200 PLHOOK	DAD 103661	! PLOTTER HOOK
4210 PLIST.	DAD 6344	! 'PLIST' STATEMENT
4220 PLOT.	DAD 64652	! 'PLOT' STATEMENT
4230 PLOTSY	DAD 100151	! PLOTTER ON/OFF FLAG
4240 POS.	DAD 4227	! POS FUNCTION RUNTIME CODE
4250 PPOLL	DAD 177423	! PARALLEL POLL REG
4260 PRARR$	DAD 70730	! PRINT# STRING ARRAY TO DISC FILE
4270 PRARR.	DAD 70167	! PRINT# NUMERIC ARRAY TO DISC FILE
4280 PRDRVR	DAD 73023	! PRINTER DRIVER ROUTINE
4290 PRDVF+	DAD 103550	! SPECIAL CHARACTER FLAG FOR LIST TIME
4300 PREOL.	DAD 70464	! PRINT# END OF LINE (DUMP BUFFER)
4310 PRINT.	DAD 71332	! SET SELECT CODE TO PRINTER IS DEVICE
4320 PRLINE	DAD 71641	! PRINT LINE RUNTIME CODE
4330 PRNTLN	DAD 101137	! 1 BYTE PRINTER LINE LENGTH
4340 PRNTR.	DAD 75631	! PRINTER IS STATEMENT
4350 PRNUM.	DAD 67220	! PRINT# A NUMBER TO A DATA FILE
4360 PRSIDL	DAD 103733	! PARSER RAM HOOK
4370 PRSTR.	DAD 66662	! PRINT# A STRING TO A DATA FILE
4380 PRTBUF	DAD 107454	! PRINT BUFFER
4390 PRTPTR	DAD 100062	! PRINT BUFFER PTR
4400 PS.C.	DAD 100222	! PRINTER SELECT CODE
4410 PTR1	DAD 177710	! I/O ADDRESSES FOR EMC POINTERS
4420 PTR1+	DAD 177712	! I/O ADDRESSES FOR EMC POINTERS
4430 PTR1-	DAD 177711	! I/O ADDRESSES FOR EMC POINTERS

Name	Address	Description
PTR1-+	DAD 177713	! I/O ADDRESSES FOR EMC POINTERS
PTR2	DAD 177714	! I/O ADDRESSES FOR EMC POINTERS
PTR2+	DAD 177716	! I/O ADDRESSES FOR EMC POINTERS
PTR2-	DAD 177715	! I/O ADDRESSES FOR EMC POINTERS
PTR2-+	DAD 177717	! I/O ADDRESSES FOR EMC POINTERS
R60+10	DAD 60010	! ROM ERROR MESSAGES
R60+12	DAD 60012	! ROM INITIALIZATION
R60+14	DAD 60014	! TEST INITIALIZATION
R60+2	DAD 60002	! ROM RUNTIME POINTERS
R60+4	DAD 60004	! ROM ASCII TABLE
R60+6	DAD 60006	! ROM PARSE TABLE
R60K	DAD 60000	! FIRST ADDRESS FOR ROMS
RAD.	DAD 62267	! PUT COMPUTER IN RADIANS TRIG MODE
RAD10	DAD 54472	! DEGREES TO RADIANS CONVERSION
RAID+1	DAD 103307	! USED BY INTERRUPT SERVICE ROUTINES
RAID+2	DAD 103310	! USED BY INTERRUPT SERVICE ROUTINES
RDARR$	DAD 70312	! READ# A STRING ARRAY FROM DISK FILE
RDARR.	DAD 70106	! READ# A NUMERIC ARRAY FROM DISK FILE
RDNUM.	DAD 67503	! READ# A NUMBER FROM DISK FILE
RDSTR.	DAD 67314	! READ# ; STRING
READ.	DAD 66221	! READ# POINTER POSITIONING
RECBUF	DAD 102600	! DISK BUFFER 400 BYTES (256 DECIMAL)
REFNUM	DAD 27530	! PARSE A NUMERIC VARIABLE REFERENCE
RELMEM	DAD 31777	! RELEASE TEMPORARY MEMORY
REM10	DAD 52533	! 'RMD' FUNCTION (REMAINDER)
RESET.	DAD 5407	! RESET KEY RUNTIME CODE
RESMEM	DAD 31741	! RESERVE SOME TEMPORARY MEMORY
RESULT	DAD 100070	! LAST CALCULATOR MODE RESULT
RETRHI	DAD 13234	! WAIT FOR RETRACE HIGH FROM CRT
RMEM	DAD 105343	! RESERVED MEMORY COUNT
RMIDLE	DAD 103706	! EXEC LOOP RAM HOOK
RND10	DAD 53741	! RND FUNCTION (GET A RANDOM NUMBER)
RNDIZ.	DAD 55713	! RANDOMIZE COMMAND
ROMEND	DAD 104145	! END OF ROM TABLE ENTRIES
ROMFL	DAD 104065	! ROM FLAG FOR INITIALIZATION ROUTINES
ROMINI	DAD 6055	! CALL BPGM'S AND ROM'S INIT ROUTINES
ROMJSB	DAD 6223	! JSB TO A BANK SELECTABLE ROM
ROMLST	DAD 104143	! LAST ENTRY IN ROM TABLE
ROMRTN	DAD 6207	! RE-SELECT ROM 0 AND RETURN
ROMTAB	DAD 104105	! BASE OF ROM TABLE
RPLOT.	DAD 64666	! 'RPLOT' STATEMENT
RSELEC	DAD 177430	! BANK SELECTABLE ROM SELECTION ADDRESS
RSTREG	DAD 22346	! RESTORE REGISTERS
RSUM8K	DAD 37670	! DO A CHECKSUM ON 8K OF MEMORY
RTCUR.	DAD 13651	! MOVE CURSOR RIGHT ON CURRENT SCREEN
RTCURS	DAD 13765	! MOVE CURSOR RIGHT IN ALPHA MEMORY
RTNSTK	DAD 100033	! TOP OF GOSUB RETURN STAK
RULITE	DAD 177704	! RUN LIGHT I/O ADDRESS
S10	DAD 103367	! FOR SAVING R10-11 DURING INTERRUPT SVC
SAD1	DAD 13723	! SET CRT ALPHA START ADDRESS
SAVER6	DAD 104066	! DISK BAIL OUT STACK POINTER FOR ERRORS
SAVR0	DAD 103200	! SYSTEM MONITOR REGISTER SAVE AREA

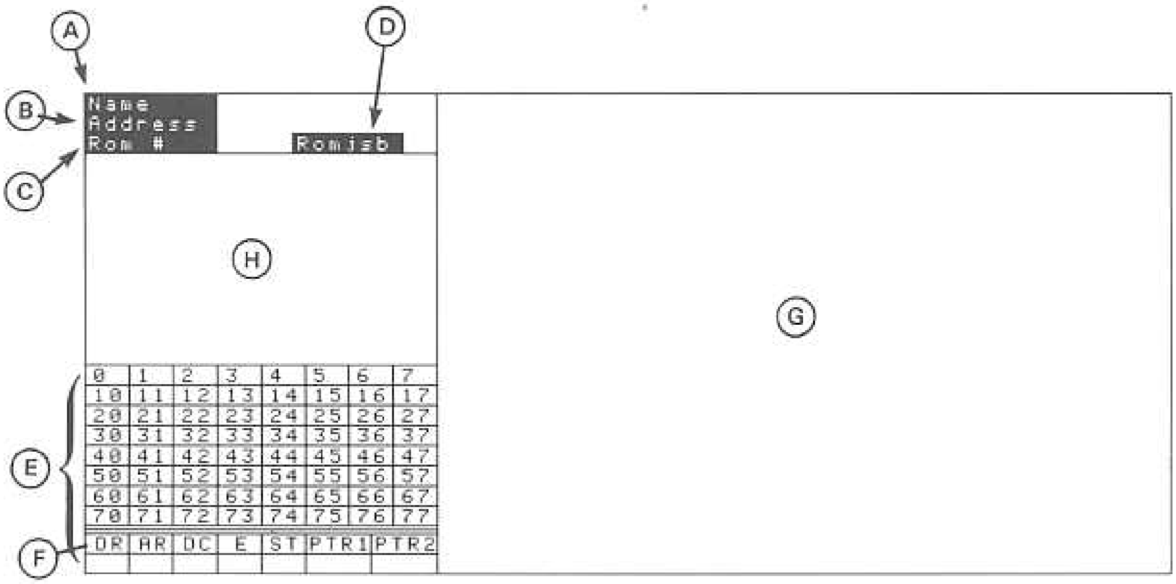
```
    Name        Address                         Description

4960 SAVR10    DAD 104063       ! R10 SAVE FOR PARSE ERRORS
4970 SAVREG    DAD 22310        ! SAVE REGISTERS ON R6
4980 SC10+1    DAD 177540       ! I/O CARD STUFF
4990 SCAN      DAD 21110        ! GET NEXT TOKEN TO R14 AT PARSE TIME
5000 SCAN+     DAD 21105        ! GCHAR AND SCAN
5010 SCRAT.    DAD 5601         ! 'SCRATCH' RUNTIME CODE
5020 SCRDN     DAD 13671        ! SCROLL DOWN THE CRT
5030 SCRUP     DAD 13736        ! SCROLL UP THE CRT
5040 SCTEMP    DAD 101721       ! S.C. TEMP STORE
5050 SEC10     DAD 54260        ! SECANT RUNTIME CODE
5060 SEMIC$    DAD 72155        ! PRINT STRING;
5070 SEMIC.    DAD 72274        ! PRINT NUMBER;
5080 SEQNO     DAD 30426        ! PARSE A LINE NUMBER
5090 SEQNO+    DAD 30422        ! PARSE A LINE NUMBER
5100 SERPOL    DAD 177425       ! MY LISTEN ADDRESS
5110 SET240    DAD 21071        ! SET THE IMMEDIATE BREAK BITS IN R17
5120 SGN5      DAD 54202        ! SGN FUNCTION
5130 SIN10     DAD 54343        ! SIN FUNCTION
5140 SKYTXT    DAD 106610       ! CALC SOFTKEYS TEXT (14*30)
5150 SPAR0     DAD 104011       ! SPARE INTERRUPT RAM HOOK (SYS MONITOR)
5160 SPAR1     DAD 104022       ! SPARE INTERRUPT RAM HOOK (UNUSED)
5170 SPECIF    DAD 103527       ! DISC VOLUME NAME
5180 SPTR1     DAD 103300       ! SYSTEM MONITOR SAVE PTR1 AREA
5190 SPTR2     DAD 103303       ! SYSTEM MONITOR SAVE PTR2 AREA
5200 SQR5      DAD 53237        ! SQUARE ROOT FUNCTION
5210 ST240+    DAD 21067        ! CLEAR R16 AND SET240
5220 STACK     DAD 102070       ! R6 STACK 500 OCTAL BYTES (320 DECIMAL)
5230 STBEEP    DAD 10441        ! STANDARD BEEP
5240 STOST     DAD 46472        ! STORE STRING ROUTINE
5250 STOSV     DAD 46057        ! STORE SIMPLE VARIABLE
5260 STRANG    DAD 103715       ! STRANGE PARAMETER TYPES INTERCEPT HOOK
5270 STRCON    DAD 24201        ! PARSE A STRING CONSTANT
5280 STREX+    DAD 23721        ! SCAN AND PARSE A STRING EXPRESSION
5290 STREXP    DAD 23724        ! PARSE A STRING EXPRESSION
5300 STRREF    DAD 24056        ! PARSE A STRING VARIABLE REFERENCE
5310 STSIZE    DAD 101741       ! STATEMENT SIZE PLACE HOLDER POINTER
5320 SUB10     DAD 52734        ! SUBTRACT TWO REAL NUMBERS IN R40 AND R50
5330 SUBROI    DAD 52724        ! SUBTRACT 2 REAL OR INTEGERS ON STACK
5340 SVCWRD    DAD 100162       ! SERVICE WORD
5350 SYSDIS    DAD 177707       ! SOS CARD ROM DISABLE ADDRESS
5360 TAN10     DAD 54363        ! TANGENT FUNCTION
5370 TIME      DAD 101123       ! TIME OF DAY
5380 TIME.     DAD 66211        ! TIME OF DAY FUNCTION
5390 TOS       DAD 101744       ! TOP R12 STAK
5400 TWOB      DAD 56760        ! GET TWO BINARY NUMBERS OFF STACK
5410 TWOR      DAD 57020        ! GET 2 REAL NUMBERS OFF R12 STACK
5420 TWOROI    DAD 57050        ! GET 2 REAL OR INTEGERS OFF R12 STACK
5430 UNBAS1    DAD 103430       ! UNUSED ROM STOLEN RAM BASE ADDRESS
5440 UNBAS2    DAD 103434       ! UNUSED ROM STOLEN RAM BASE ADDRESS
5450 UNEQ$.    DAD 3603         ! COMPARE STRINGS FOR UNEQUAL
5460 UNEQ.     DAD 62632        ! COMPARE NUMBERS FOR UNEQUAL
5470 UNQUOT    DAD 24366        ! PARSE AN UNQUOTED STRING
```

8.3 System Operation and Routines

This section provides documentation for certain areas of system operation. It also shows the input conditions required and the outputs produced by selected system routines. The names and addresses of the system routines detailed here are also on the disc.

The system routines are arranged in alphabetical order. Their area of primary use is noted. Because a routine is listed under a certain application does not limit its use to that area. For example, many utility routines may also be used during run time operations.

The format of the individual system routines is shown here:



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

B. Address: Permanent octal address of the routine in computer memory.

C. ROM#: The ROM that must be selected if this routine needs to be called through ROMJSB.

D. ROM #: The "Y" or "N" entry indicates if this routine needs to to be called through ROMJSB.

E. Registers: Shaded areas indicate registers used by this routine.

F. DR,AR,DC,E,ST,PTR1,PTR2: Entries in these boxes indicate exit conditions of this routine. The following symbols are used:

Symbol	Meaning
-	Unchanged.
U	Unknown.
*	Refer to the description (G).

G. Conditions: When applicable, shows input and output stack contents, and output register contents.

H. Description: Contains description of routine.

ABS5
MATH

Name	ABS5
Address	54525
Rom #	0 **Romjsb** N

System function that returns the absolute value of a number.

(Refer to the system function ABS in the owner's manual.)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
       Argument (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
       Absolute value of argument (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of absolute value.

R60-R67 = Copy of original argument value.

ADD10
MATH

Name	ADD10
Address	53030
Rom #	0 **Romjsb** N

Adds two real (floating-point) numbers.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT REGISTER CONTENTS

R40-R47 = Real value A (8-bytes)
R50-R57 = Real value B (8-bytes)

OUTPUT STACK CONTENTS

```
       Result  A+B (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of result A+B

NOTE: The two numbers must be in floating-point format and the CPU must be in BCD mode when ADD10 is called or the result will be incorrect.

ADDROI
MATH

Name	ADDROI
Address	52745
Rom #	0 **Romjsb** N

Adds two real or tagged-integer numbers.

(This is the main runtime entry point for the system operator +.)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
       Real or tagged-integer A (8-bytes)
       Real or tagged-integer B (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
       Result A+B (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of the result

NOTE: The result may be either a real or a tagged-integer number. The CPU must be in BCD mode before calling ADDROI.

ALFA
PARSE

Name	ALFA
Address	21656
Rom #	0 **Romjsb** N

Checks the character in R20 to see if it's between 'A' and 'Z' or 'a' and 'z'. If it's lower case, it's shifted to upper case.

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

DR	AR	DC	E	ST	PTR1	PTR2
20	U	B	‡	U	-	-

INPUT CONDITIONS

R20 = The character

OUTPUT CONDITIONS

R20 = The character (shifted to upper case if it was lower case)

E = 0 If it was not an alpha character
E = 1 If it was an alpha character

ALFAL.
CRT

Name	ALFAL.
Address	12466
Rom #	0 **Romjsb** N

Forces ALPHA ALL mode on the CRT.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	-	U	-	-

OUTPUT CONDITIONS

The CRT will be in ALPHA ALL mode.
The actual code for ALFAL. is:

```
ALFAL.   BIN
         LDBD R30,=CRTSTS  !GET CRT STATUS
         LLB R30           !GRAPH/GRAPHALL?
         ELB R30           !
         ERB R30           !
         JCY ALFAL1        !JIF YES
         JNG ARTN          !JIF ALPHA ALL
ALFAL1   JSB =CRTWPO       !BLANK CRT
         LDB R30,=100      !GET 'ALL' MASK
         LDBD R31,=CRTSTS  !GET STATUS
         ANM R31,=177      !TRASH GRAPH BIT
         ORB R31,R30       !OR IN 'ALL' BIT
         STBD R31,=CRTSTS  !SET CRT STATUS
         LDM R36,=300,77   !LOAD 37700
         STMD R36,=ASIZE   !SET ALPHA SIZE
         JSB =CRTBLK       !CLEAR MEMORY
         JSB =CURS         !OUTPUT CURSOR
         JMP CRTUNW        !UNBLANK CRT
```

ALPHA.
CRT

Name	ALPHA.
Address	12413
Rom #	0 **Romjsb** N

Forces the CRT to the ALPHA NORMAL mode. If it was in ALPHA ALL or GRAPH ALL mode, then the CRT memory is initialized by calling CRTINT.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

OUTPUT CONDITIONS

The CRT display will be in ALPHA NORMAL.

The CRT memory will have been initialized and top of stack set equal to R12-R13 if the display was in ALPHA ALL or GRAPH ALL mode at entry.

ALPHA.
CRT

Name	ALPHA
Address	12542
Rom #	0 Romjsb N

If the CRT display is in GRAPH NORMAL mode at entry, it will be switched to ALPHA NORMAL mode, else nothing will be done.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

OUTPUT CONDITIONS

If the CRT is in ALPHA NORMAL mode at entry, it will be in ALPHA NORMAL mode at exit. If the CRT is in ALPHA ALL mode at entry, it will be in ALPHA ALL mode at exit. If the CRT is in GRAPH NORMAL mode at entry, it will be in ALPHA NORMAL mode at exit. If the CRT is in GRAPH ALL mode at entry, it will be in GRAPH ALL mode at exit. One return address will also be thrown away before returning if it was in GRAPH ALL mode, so it won't return to the calling routine.

ASSIG.
DISC

Name	ASSIG.
Address	65466
Rom #	320 Romjsb Y

Runtime code for the BASIC statement.

ASSIGN#

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

INPUT STACK CONTENTS

```
        Buffer number (8 bytes)
        File name length (2 bytes)
        File name address (3 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
R12---->
```

ATN2.
MATH

Name	ATN2.
Address	77157
Rom #	0 Romjsb Y

Performs the system function ATN2(Y,X) which returns the arctangent of Y/X in the proper quadrant.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	D	U	U	-	-

INPUT STACK CONTENTS

```
        Y-value (8 bytes)
        X-value (8 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        ATN2(Y,X) (8 bytes)
R12---->
```

BEEP.
MISC.

Name	BEEP.
Address	10361
Rom #	0
Romjsb	Y

Runtime code for the BEEP statement.

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**

DR	AR	DC	E	ST	PTR1	PTR2
31	U	U	U	U	-	-

INPUT CONDITIONS

Top of stack and R12 are compared to see if there are any optional parameters on the R12 stack. If none, then a JMP is made to STBEEP.

BEEP A,B would make the stack look like this:

```
          A (8 bytes)
          B (8 bytes)
  R12---->
```

BKSPC
CRT

Name	BKSPC
Address	11520
Rom #	0
Romjsb	N

Does a backspace. (Same as if the backspace key had been pressed.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
32	24	B	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the same address as the CRT controllers byte address register (CRTBAD).

The cursor must be off at entry (a call must have been made to DECUR2).

BLKLIN
CRT

Name	BLKLIN
Address	14165
Rom #	0
Romjsb	N

Fills from current CRT byte address to the end of the line with carriage return characters (15 octal).

Alters CRTBYT, leaving it pointing to the start of the next line.

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**

DR	AR	DC	E	ST	PTR1	PTR2
66	76	B	-	U	-	-

INPUT CONDITIONS

The CRT byte address pointer (CRTBAD) must be pointing to the address where blanking is to start.

OUTPUT CONDITIONS

The CRT byte address pointer will be pointing to the first character of the next line.

The actual code for BLKLIN is:

```
BLKLIN   BIN
         JSB  =COLUMN
         LDB  R32,=15
L8       JSB  =OUTCHI
         ICB  R66
         CMB  R66,=80D
         JNZ  L8
         RTN
```

BPINI
MISC.

Name	BPINI		
Address	6113		
Rom #	0	**Romjsb**	Y

Calls the INIT routines in all of the binary programs present in memory.

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**

DR	AR	DC	E	ST	PTR1	PTR2
0	U	U	U	U	U	U

INPUT CONDITIONS

ROMFL = Reason for the call:

```
      0 Power on
      1 Reset
      2 Scratch
      3 Loadbin
      4 Run,Init
      5 Load
      6 Stop,Pause
      7 Chain
     10 Allocate class >56
     11 De-allocate class >56
     12 De-compile class >56
     13 Program halt on error
```

NOTE: Binary programs must insure that R0 does not get destroyed during their INIT routine as R0 is used by BPINI as a counter of which binary program is next.

BYTCRT
CRT

Name	BYTCRT		
Address	14004		
Rom #	0	**Romjsb**	N

Sets the byte address in CRTBYT and sends it to the CRT controller (CRTBAD).

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

DR	AR	DC	E	ST	PTR1	PTR2
-	-	-	-	U	-	-

INPUT REGISTER CONTENTS

The register pair pointed to by the DRP must contain the address to be stored to CRTBYT and CRTRAM.

The actual code is:

```
BYTCRT  STMD R#,=CRTBYT
        SAD
        JSB =CHKSTS
        PAD
        STMD R#,=CRTBAD
        RTN
```

CEIL10
MATH

Name	CEIL10		
Address	54412		
Rom #	0	**Romjsb**	N

Runtime code for the system function CEIL.

Returns the smallest integer >= X.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
       X-value (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
       CEIL(X) result (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of result

CHKSTS
CRT

Name	CHKSTS
Address	13204
Rom #	0
Romjsb	N

Waits for the CRT controller to be not busy. (When CHKSTS returns it is safe to store to CRTBAD or CRTDAT.)

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

DR	AR	DC	E	ST	PTR1	PTR2
30	-	B	-	U	-	-

This is the actual code for CHKSTS:

```
    CHKSTS   BIN
             DRP R30
    BUSY     LDBD R#,=CRTSTS    !GET CRT STATUS
             JOD BUSY           !LOOP IF BUSY
             RTN                !ELSE RETURN
```

This routine is useful when you want to store directly to CRTBAD and/or CRTDAT (such as when doing high speed graphics or alpha displays).

CHSROI
MATH

Name	CHSROI
Address	52672
Rom #	0
Romjsb	N

Changes the sign of a real or integer number.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	-	U	U	-	-

INPUT STACK CONTENTS

```
         Real or tagged-integer  (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
         Real or tagged-integer  (8-bytes)
  R12---->
```

The actual code is:

```
     CHSROI    POMD R40,-R12
               CMB R44,=377
               JCY CHS10X
               TSM R40
               JZR CHS11
               LLB R41
               NCB R41
               ERB R41
     CHS11     PUMD R40,+R12
               RTN
     CHS10X    TCM R45
               JMP CHS11
```

CLEAR.
CRT

Name	CLEAR.
Address	14225
Rom #	0
Romjsb	N

Does the same thing as a CLEAR statement in BASIC.

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**

DR	AR	DC	E	ST	PTR1	PTR2
31	U	B	-	U	-	-

This is the runtime entry point for the BASIC reserved word CLEAR.

CLREOL
CRT

Name	CLREOL
Address	13447
Rom #	0
Romjsb	N

Clears to the end of the current ALPHA line (based upon the contents of CRTBYT) and leaves CRTBYT pointing to where it was.

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**

DR	AR	DC	E	ST	PTR1	PTR2
66	6	B	-	U	-	-

INPUT CONDITIONS

CRTBYT = Current ALPHA cursor location.

CNTRTR
CRT

Name	CNTRTR
Address	13245
Rom #	0
Romjsb	N

Counts a specified number of CRT retrace periods.

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

DR	AR	DC	E	ST	PTR1	PTR2
30	-	-	-	U	-	-

This routine can be used when delays of 16.67 milliseconds to 4.27 seconds are desired (in steps of multiples of 16.67 milliseconds). The CRT controller refreshes the CRT 60 times a second. That means there is a retrace period every 1/60 of a second or every 16.67 milliseconds. This routine simply counts the number of retrace periods specified when it is called.

INPUT REGISTER CONTENTS
R30 = Number of retraces to be counted

OUTPUT REGISTER CONTENTS
R30 = 0

NOTE: The CPU must be in BIN mode before this routine is called

R30 is a one-byte count, thus limiting the count to 256. (If R30 is 0 at entry CNTRTR will count for 256 retraces.)

COLUMN
CRT

Name	COLUMN
Address	14206
Rom #	0
Romjsb	N

Calculates the column number of the current cursor location on ALPHA display. It is returned as a number between 0 and 117 (octal).

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**

DR	AR	DC	E	ST	PTR1	PTR2
66	76	-	-	U	-	-

INPUT CONDITIONS

CRTBYT must contain the current cursor address.

OUTPUT REGISTER CONTENTS

R66-R67 = Column number (0-117)
R76-R77 = 120 (octal)

NOTES:
The CPU must be in BIN mode at entry.
The actual code is:

```
COLUMN  LDM R76,=120,0       !LINE LEN=80
        LDMD R66,=CRTBYT     !CURSOR ADDR.
        ARP R76              !SAVE CYCLES
MOD     SBM R#,R#            !SUBTRACT UNTIL
        JCY MOD              !   <0
        ADM R#,R#            !MAKE POSITIVE
        RTN                  !DONE
```

COMMA$
PRINT

Name	COMMA$
Address	72146
Rom #	0
Romjsb	Y

Prints a string to the print or display buffer. Same as:

PRINT "ABC",

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

        Length of string     (2 bytes)
        Address of string    (3 bytes)
  R12----> *

OUTPUT STACK CONTENTS

        (empty)
  R12---->

 NOTE: DISP. or PRINT. must be called
 prior to calling COMMA$ to set up the
 select code and buffer pointers.
```

COMMA.
PRINT

Name	COMMA.
Address	72265
Rom #	0
Romjsb	Y

Prints a number to the print or display buffer. Same as:

PRINT 34,

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

        Number to be printed (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

 NOTE: DISP. or PRINT. must be called
 prior to calling COMMA. to set up the
 select code and buffer pointers.
```

CONBI3
MATH

Name	CONBI3
Address	4516
Rom #	0
Romjsb	N

Converts a 23-bit signed binary number to a floating-point value.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

```
INPUT REGISTER CONTENTS

  R55-R57 = 23-bit signed binary number.
        The 23 least significant bits are
        the magnitude (0 TO 2^23-1) and the
        most significant bit is the sign
        (0=positive and 1=negative) giving
        a range of -8388607 TO +8388607

OUTPUT REGISTER CONTENTS

  R40-R47 = The equivalent floating-point
                value.
```

CONBIN
MATH

Name	CONBIN
Address	4401
Rom #	0
Romjsb	N

Converts a 15-bit signed binary number to a floating-point value.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

INPUT REGISTER CONTENTS

```
R36-R37 = 15-bit signed binary number.
        The 15 least significant bits are
        the magnitude (0-32767) and the
        most significant bit is the sign
        (0=positive and 1=negative) giving
        a range of -32767 to +32767.
```

OUTPUT REGISTER CONTENTS

```
R40-R47 = The equivalent floating-point
              value.
```

CONCA.
MISC.

Name	CONCA.
Address	76366
Rom #	0
Romjsb	T

Concatenates two strings.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        A$ Length      (2 bytes)
        A$ Address     (3 bytes)
        B$ Length      (2 bytes)
        B$ Address     (3 bytes)
   R12---->
```

OUTPUT STACK CONTENTS

```
        A$ & B$ Length  (2 bytes)
        A$ & B$ Address (3 bytes)
   R12---->
```

CONINT
MATH

Name	CONINT
Address	45116
Rom #	0
Romjsb	N

Converts a floating-point value to a 16-bit unsigned value with a separate sign flag.

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**

DR	AR	DC	E	ST	PTR1	PTR2
-	-	-	U	-	-	-

INPUT REGISTER CONTENTS

```
R60-R67 = Floating-point value
```

OUTPUT REGISTER CONTENTS

```
R76-R77 = 16-bit unsigned binary value
R32     = Sign of value
            If R32=0 then value is positive
            If R32#0 then value is negative
```

NOTE: This routine doesn't check to insure that R60-R67 contains a floating point number, so if it contains a tagged-integer or some other garbage, you'll get indeterminate results.

CONINT does a SAD at entry and a PAD at exit, so all status is preserved (not including the E register).

CONTR.
MISC.

Name	CONTR.
Address	61620
Rom #	1
Romjsb	Y

Runtime code for the SET I/O statement.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

          I/O card number   (8 bytes)
          Register number   (8 bytes)
          Control value     (8 bytes)
   R12---->

OUTPUT STACK CONTENTS

          (empty)
   R12---->
```

COS10
MATH

Name	COS10
Address	54353
Rom #	0
Romjsb	N

Returns the COS(X)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

          X value     (8-bytes)
   R12---->

OUTPUT STACK CONTENTS

          COS(X) value    (8-bytes)
   R12---->
```

COT10
MATH

Name	COT10
Address	54333
Rom #	0
Romjsb	N

Returns the COT(X)

(cotangent)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

          X value     (8-bytes)
   R12---->

OUTPUT STACK CONTENTS

          COT(X) value    (8-bytes)
   R12---->
```

COUNTK
MISC.

Name	COUNTK
Address	14411
Rom #	0 **Romjsb** N

Used for repeating keys.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	-	U	-	-

If a key is not pressed at entry or if it is released before the key repeat wait is done then a call is made to EOJ2, else the key repeat speed is forced to KRPET2 and the service request flags are set in R17 and SVCWRD.

CRT.
CRT

Name	CRT.
Address	57307
Rom #	1 **Romjsb** Y

Performs the CRT IS statement.

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**

DR	AR	DC	E	ST	PTR1	PTR2
65	U	U	U	U	-	-

INPUT STACK CONTENTS

```
  Top of stack-> Select code (8-bytes)
        Optional line length (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

CRTBLK
CRT

Name	CRTBLK
Address	12246
Rom #	0 **Romjsb** N

Fills all of ALPHA memory with carriage return characters (octal 15).

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

DR	AR	DC	E	ST	PTR1	PTR2
36	30	B	-	U	-	-

OUTPUT CONDITIONS

The CRT will be in the blanked-out mode (that is, CRTUNW will have to be called after CRTBLK).

```
R32     = 15
R36-R37 = 0
```

CRTINT
CRT

Name	CRTINT
Address	12176
Rom #	0 Romjsb N

Initializes the CRT memory to the ALPHA NORMAL mode with all of ALPHA and GRAPH memory cleared.

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

DR	AR	DC	E	ST	PTR1	PTR2
31	30	B	U	U	-	-

OUTPUT CONDITIONS

R36-R37 = 0

The top of stack will have to be set equal to R12-R13. May GRAPH parameters will be set to the default values.

The CRT start address (CRTSAD) and the CRT byte address (CRTSAD) will be set to 0.

CRTPOF
CRT

Name	CRTPOF
Address	12334
Rom #	0 Romjsb N

Turns off the CRT's high voltage supply. Would be used if a device were hooked up to the computer that needed excessive power and didn't need to be run at the same time as the CRT display.

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

DR	AR	DC	E	ST	PTR1	PTR2
31	30	B	-	U	-	-

The code for CRTPOF is:

```
CRTPOF   LDB R30,=6
         BIN
         JSB =RETRHI
         LDBD R#,=CRTSTS
         ORB R#,R30
         STBD R#,=CRTSTS
         RTN
```

CRTPUP
CRT

Name	CRTPUP
Address	12341
Rom #	0 Romjsb N

Turns on the CRT's high voltage supply. Would be used if a device were hooked up to the computer that needed excessive power and didn't need to be run at the same time as the CRT display.

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

DR	AR	DC	E	ST	PTR1	PTR2
31	-	-	-	U	-	-

The code for CRTPUP is:

```
CRTPUP   LDBD R31,=CRTSTS
         ANM R31,=373
         STBD R31,=CRTSTS
         LDB R30,=3
         JSB =CNTRIR
         JMP CRTUNW
```

The CPU must be in BIN mode at entry.

CRTUNW
CRT

Name	CRTUNW
Address	12360
Rom #	0 Romjsb N

Un-blanks the CRT display (starts the electron beam scanning)

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

DR	AR	DC	E	ST	PTR1	PTR2
31	-	-	-	U	-	-

The CRT may need to be blanked for two reasons:

1) If the CRT controller doesn't need to be refreshing the display, the CRT memory can be accessed much more quickly than during refreshing.
2) When switching CRT modes ugly flashes can be seen if the CRT is not blanked first.

The actual code for CRTUNW is:

```
CRTUNW   JSB  =RETPHI
         LDBD R#,=CRTSTS
         ANM  R#,=371
         STBD R#,=CRTSTS
         RTN
```

CRTWPO
CRT

Name	CRTWPO
Address	12374
Rom #	0 Romjsb N

Blanks the CRT display (stops the electron beam scanning)

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

DR	AR	DC	E	ST	PTR1	PTR2
31	30	0	-	U	-	-

The CRT may need to be blanked for two reasons:

1) If the CRT controller doesn't need to be refreshing the display, the CRT memory can be accessed much more quickly than during refreshing.
2) When switching CRT modes ugly flashes can be seen if the CRT is not blanked first.

The actual code for CRTWPO is:

```
CRTWPO   LDB  R30,=2
         BIN
         JSB  =RETPHI
         LDBD R#,=CRTSTS
         ORB  R#,R30
         STBD R#,=CRTSTS
         RTN
```

CSEC10
MATH

Name	CSEC10
Address	54300
Rom #	0 Romjsb N

Returns the CSC(X)

(cosecant)

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**

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
              X value (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
              CSC(X) value (8-bytes)
  R12---->
```

CSIZE.
CRT

Name	CSIZE.
Address	66570
Rom #	1 **Romjsb** Y

Runtime code for the BASIC statement

CSIZE X

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        CSIZE       value (8 bytes)
  R12---->
                 *
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

CURS
CRT

Name	CURS
Address	14030
Rom #	0 **Romjsb** N

Outputs a cursor at the current CRTBYT location.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	-	U	-	-

INPUT CONDITIONS

CRTBYT and CRTBAD must point to the location at which the cursor is to be output.

NOTE: The cursor is created by setting the most significant bit of the character at the cursor location. This causes that character to be highlighted, or shown in inverse video. If the character at that location already has its MSB set, then the bit is cleared. DECUR2 does just the opposite. In this way the cursor is not destructive when it moves through inverse video fields. There is a flag (CURSON) which tells CURS and DECUR2 if the cursor is already on so that the MSB of the character will not be erroneously toggled if successive calls to CURS or DECUR2 are made.

CVNUM
PRINT

Name	CVNUM
Address	72401
Rom #	0 **Romjsb** Y

Formats a floating-point number into ASCII characters for printing or display-ing.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

INPUT REGISTER CONTENTS

```
R30-R31 = Pointer to output buffer
R40-R47 = Floating-point number to be
          formatted into output buffer.
```

OUTPUT REGISTER CONTENTS

```
R30-R31 = Pointer to next available byte
          in the output buffer.
```

DALLOC
MISC.

Name	DALLOC
Address	47123
Rom #	0
Romjsb	Y

Totally de-allocates the current BASIC program.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	U

INPUT CONDITIONS

R16 must be even. If it is odd, DALLOC will return without doing anything.

DATE.
MISC.

Name	DATE.
Address	32073
Rom #	0
Romjsb	N

Returns the DATE.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	-	-	U	-	-

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of date

OUTPUT STACK CONTENTS

The date (8 bytes)
R12---->

NOTE: DATE will always return as a tagged integer.

DCLIN#
PRINT

Name	DCLIN#
Address	34607
Rom #	0
Romjsb	N

Decompiles a BASIC program line number.

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

DR	AR	DC	E	ST	PTR1	PTR2
57	30	B	U	U	-	-

INPUT CONDITIONS

R30-R31 = Pointer to output buffer
R65-R67 = Line number as 5 BCD digits

OUTPUT CONDITIONS

R30-R31 = Pointer to output buffer (after the line number was pushed out as ASCII characters).

DCSLOP
MISC.

Name	DCSLOP
Address	35132
Rom #	0 Romjsb N

Reverses a string of bytes into the lower 64K address space.

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**

DR	AR	DC	E	ST	PTR1	PTR2
76	30	-	-	U	-	U

INPUT CONDITIONS

```
Assumes CPU is in BIN mode at entry.
R30-R31 = First byte sink (lowest)
R76-R77 = Number of bytes to move
PTR2    = First byte + 1 of source
                  (highest)
```

The actual code is:

```
DCSLOP    LDBI R75,=PTR2-
          PUBD R75,+R30
          DCM R76
          JNZ DCSLOP
          RTN
```

DECUR2
CRT

Name	DECUR2
Address	13467
Rom #	0 Romjsb N

Erases the cursor at the current CRTBYT location.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	-	U	-	-

INPUT CONDITIONS

CRTBYT and CRTBAD must point to the location of the cursor.

NOTE: The cursor is created by setting the most significant bit of the character at the cursor location. This causes that character to be highlighted, or shown in inverse video. If the character at that location already has its MSB set, then the bit is cleared. DECUR2 does just the opposite. In this way the cursor is not destructive when it moves through inverse video fields. There is a flag (CURSON) which tells CURS and DECUR2 if the cursor is already on so that the MSB of the character will not be erroneously toggled if successive calls to CURS and DECUR2 are made.

DEFA+.
MATH

Name	DEFA+.
Address	61576
Rom #	0 Romjsb Y

Turns the system math defaults on.

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

DR	AR	DC	E	ST	PTR1	PTR2
36	-	-	-	U	-	-

OUTPUT REGISTER CONTENTS

R36 = 1

The actual code is:

```
DEFA+.    CLB R36
          ICB R36
          JMP STORDF
          BYT 241
DEFA-.    CLB R36
STORDF    STBD R#,=DEFAUL
          RTN
```

DEFA-.
MATH

Name	DEFA-.
Address	61604
Rom #	0 Romjsb Y

Turns the system math defaults off.

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

DR	AR	DC	E	ST	PTR1	PTR2
36	-	-	-	U	-	-

OUTPUT REGISTER CONTENTS

R36 = 0

The actual code is:

```
DEFA+.     CLB R36
           ICB R36
           JMP STORDF
           BYT 241
DEFA-.     CLB R36
STORDF     STBD R#,=DEFAUL
           RTN
```

DEG.
MATH

Name	DEG.
Address	62257
Rom #	0 Romjsb Y

Puts the computer in degrees mode for math operations.

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

DR	AR	DC	E	ST	PTR1	PTR2
36	-	-	-	U	-	-

The actual code is:

```
DEG.       LDB R36,=90C
STODRG     STBD R#,=DRG
           RTN
           BYT 241
RAD.       CLB R36
           JMP STODRG
           BYT 241
GRAD.      CLB R36
           DCB R36
           JMP STODRG
```

DEG10
MATH

Name	DEG10
Address	54736
Rom #	0 Romjsb N

Runtime code for the system function RTD.

Converts radians to degrees.

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**

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
          Radian-value    (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
          Degrees result   (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of result

DIGIT
PARSE

Name	DIGIT
Address	21710
Rom #	0
Romjsb	N

Checks a character to see if it's a digit (0-9, ASCII codes 60-71 octal).

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

DR	AR	DC	E	ST	PTR1	PTR2
-	ar	-	*	-	-	-

INPUT CONDITIONS

R20 = The character

OUTPUT CONDITIONS

R20 = The character

E = 0 If it was not a digit
E = 1 If it was a digit

DISP.
CRT

Name	DISP.
Address	71311
Rom #	0
Romjsb	Y

Sets up SCTEMP so that it contains the current CRT IS select code. It's usually used prior to calling DRV12.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	U	-	-	U	-	-

DIV10
MATH

Name	DIV10
Address	52441
Rom #	0
Romjsb	N

Divides one real number into a second real number.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS
(empty)
R12---->

INPUT REGISTER CONTENTS
R40-R47 = Real number A (8-bytes)
R50-R47 = Real number B (8-bytes)

OUTPUT STACK CONTENTS
Result B/A (8-bytes)
R12---->

OUTPUT REGISTER CONTENTS
R40 R47 = Copy of result B/A

NOTE: The CPU must be in BCD mode before DIV10 is called.

The two arguments must be real numbers or the result will be unknown.

DIV2
MATH

Name	DIV2
Address	52436
Rom #	0 — Romjsb N

Divides one real or tagged-integer number into a second real or tagged-integer number.
(This is the main runtime entry point for the system operator /.)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        Real or tagged-integer A (8-bytes)
        Real or tagged-integer B (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        Result   A/B  (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of result

NOTE: The CPU must be in BCD mode before this routine is called.

DMNDCR
PARSE

Name	DMNDCR
Address	25175
Rom #	0 — Romjsb Y

Flags an error if R14 is not a carriage return or an @ token.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	-	U	U	-	-

INPUT CONDITIONS

R14 = Incoming token

OUTPUT CONDITIONS

If R14 didn't contain a carriage return token or an @ token then ERROR will have been called.

DNCUR.
CRT

Name	DNCUR.
Address	13607
Rom #	0 — Romjsb N

Moves the cursor down one line on the ALPHA display.
(Checks to see if it goes off of the current page of CRT memory and wraps it around if it does.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

DNCURS
CRT

Name	DNCURS
Address	13751
Rom #	0
Romjsb	N

Moves the cursor address down one line in ALPHA memory. (Doesn't check to see if it goes off of the current page of CRT memory.)

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

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

DRAW.
CRT

Name	DRAW.
Address	64727
Rom #	1
Romjsb	Y

Runtime code for the BASIC statement

DRAW X,Y

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        X-value    (8 bytes)
        Y-value    (8 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)    (0 bytes)
R12---->
```

DRV12.
PRINT

Name	DRV12.
Address	6722
Rom #	0
Romjsb	Y

PRINT and DISP output driver. Vectors the outputs to either the OUTSTR routine or the PRDRVR routine or the IOTRFC hook, based upon the value of the current select code in SCTEMP.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

A call to PRINT. or DISP. should have been performed before calling DRV12. This will set up SCTEMP to point to either the PRINTER IS device or the CRT IS device.

EMOVDN
MISC.

Name	EMOVDN
Address	32161
Rom #	0
Romjsb	N

Moves the contents of a block of memory to a location lower than its current location, starting with the lowest addressed bytes and working up to the highest.

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

DR	AR	DC	E	ST	PTR1	PTR2
50		-	-	U	*	*

INPUT CONDITIONS

The CPU must be in BIN mode at entry.
PTR1 = Last byte of source (low)
PTR2 * = Last byte of sink (low)
R45-R47 = Number of bytes to move

OUTPUT CONDITIONS

PTR1 = First byte of source +1 (high)
PTR2 = First byte of sink +1 (high)
R45-R47 = 0

NOTE: EMOVUP and EMOVDN are backwards from MOVUP and MOVDN. In other words, EMOVUP does for extended memory what MOVDN does for the lower 64K of memory and EMOVDN corresponds to MOVUP.

EMOVUP
MISC.

Name	EMOVUP
Address	32231
Rom #	0
Romjsb	N

Moves the contents of a block of memory to a location higher than its current location, starting with the highest addressed bytes and working down to the lowest.

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

DR	AR	DC	E	ST	PTR1	PTR2
50		-	-	U	*	*

INPUT CONDITIONS

The CPU must be in BIN mode at entry.
PTR1 = First byte of source +1 (high)
PTR2 = First byte of sink +1 (high)
R45-R47 = Number of bytes to move

OUTPUT CONDITIONS

PTR1 = Last byte of source (low)
PTR2 = Last byte of sink (low)
R45-R47 = 0

NOTE: EMOVUP and EMOVDN are backwards from MOVUP and MOVDN. In other words, EMOVUP does for extended memory what MOVDN does for the lower 64K of memory and EMOVDN corresponds to MOVUP.

EOJ2
MISC.

Name	EOJ2
Address	14525
Rom #	0
Romjsb	N

Turns off the key-board service request bit in SVCWRD and, if no other service requests are pending, turns off the service request bit in R17. Also, resets the key repeat speed to the value in KRPET1.

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

DR	AR	DC	E	ST	PTR1	PTR2
32	32	-	-	U	-	-

EPS10 — MATH

Name	EPS10
Address	54722
Rom #	0 **Romjsb** N

Returns the smallest positive number the computer is capable of handling (1E-499).

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

DR	AR	DC	E	ST	PTR1	PTR2
50	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        (whatever)
  R12---->

OUTPUT STACK CONTENTS

        (whatever)
        Smallest number 1E-499   (8-bytes)
  R12---->
```

EQ$. — MATH

Name	EQ$.
Address	3564
Rom #	0 **Romjsb** N

Checks two strings for equality.

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**

DR	AR	DC	E	ST	PTR1	PTR2
70	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        Length of string 'A'    (2-bytes)
        Address of string 'A'   (3-bytes)
        Length of string 'B'    (2-bytes)
        Address of string 'B'   (3-bytes)
  R12---->

OUTPUT STACK CONTENTS

        True/False value        (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R70-R77 = Copy of true/false value.

  NOTE: The true/false value is =0 if
  false, =1 if true and is in floating-
  point format.
```

EQ. — MATH

Name	EQ.
Address	62623
Rom #	0 **Romjsb** Y

Tests two numbers for equality.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

```
INPUT STACK CONTENTS

        A-value   (8-bytes)
        B-value   (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        True/False value  (8-bytes)
  R12---->

  NOTE: The true/false value will always
  be a tagged integer and will be a 1 if
  true and a 0 if false.
```

ERROR
MISC.

Name	ERROR
Address	10224
Rom #	0 — Romjsb: N

Sets flags for error reporting.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	-	0	U	-	-

```
INPUT CONDITIONS

   ERROR must be called by this format:
       *    JSB =ERROR
            BYT error number

OUTPUT CONDITIONS

   R17 has bits 6 and 7 set.
   ERRORS = error number
   ERNUM# = error number adjusted for
                external ROMs or binary programs
   ERROM# = ERRROM
   If in RUN mode (R16=2) then
                ERLIN# = error line number
   When control returns to the EXEC loop,
   these flags will cause an error message
   to be output. All registers are saved.
```

ERROR+
MISC.

Name	ERROR+
Address	10220
Rom #	0 — Romjsb: N

Sets flags for error reporting.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	-	0	U	-	-

```
INPUT CONDITIONS

   ERROR+ must be called by this format:
            JSB =ERROR+
            BYT error number

OUTPUT CONDITIONS

   R17 has bits 6 and 7 set.
   ERRORS = error number
   ERNUM# = error number adjusted for
                external ROMs or binary programs
   ERROM# = ERRROM
   If in RUN mode (R16=2) then
                ERLIN# = error line number
   When control returns to the EXEC loop,
   these flags will cause an error message
   to be output. One RTN address is
   discarded from the R6 stack before
   ERROR+ returns.  All registers are
   saved.
```

EXP5
MATH

Name	EXP5
Address	53174
Rom #	0 — Romjsb: N

Returns EXP(X)

e ^ X

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

         X value    (8-bytes)
   R12---->

OUTPUT STACK CONTENTS

         EXP(X)     (8-bytes)
   R12---->
```

Name	FASTBS		
Address	11565		
Rom #	0	**Romjsb**	N

Does a fast backspace. (Same as if the backspace key had been pressed while the shift key was held down.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
66	76	B	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the same address as the CRT controllers byte address register (CRTBAD).

The cursor must be off at entry (a call must have been made to DECUR2).

FASTBS
CRT

Name	FBPGM		
Address	50333		
Rom #	0	**Romjsb**	N

Tries to locate (in memory) a binary program having a given binary program number. If found, BINTAB will be set to its base address; otherwise, BINTAB will be set to zero.

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

DR	AR	DC	E	ST	PTR1	PTR2
20	0	-	-	U	-	-

INPUT CONDITIONS

Assumes the CPU is in BIN mode at entry.
R22 = The desired binary program #.

OUTPUT CONDITIONS

R20-R21 = Base address of the binary program if found, else zero.
BINTAB = Base address of the binary program if found, else zero.

FBPGM
MISC.

Name	FETAVA		
Address	45505		
Rom #	0	**Romjsb**	N

Fetches the address an array element (FETch Array Variable Address).

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        Ptr to variable area        (3 bytes)
        Row index                   (2 bytes)
        Col index (optional)        (2 bytes)
        Dim flag                    (1 byte)
  R12---->
OUTPUT STACK CONTENTS
        (empty)
  R12---->
OUTPUT REGISTER CONTENTS
  R20-R21 = Max len if string array
  R46     = Header byte of array
  R70-R72 = Abs. address of variable name
  R75-R77 = Abs. address of element value
  PTR2    = Address of element value
```

NOTE: At entry, 'Ptr to variable area' is an address which is relative to FWCURR. 'Col index' is present only if the array is two dimensional. 'Dim flag' is even if the array is two dimensional, odd if one.

FETAVA
MISC.

FETSVA
MISC.

Name	FETSVA
Address	45305
Rom #	0
Romjsb	N

FETch Simple Variable Address. Takes an address which is relative to FWCURR and changes it to an absolute address.

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

DR	AR	DC	E	ST	PTR1	PTR2
46	47	B	-	U	-	*

INPUT REGISTER CONTENTS

R65-R67 = Address of variable area (relative to FWCURR)

OUTPUT REGISTER CONTENTS

R46 = Header byte of variable
R70-R72 = Abs. address of variable name
PTR2 = Abs. address of least significant byte of address of variable name in variable storage area.

FLIP.
MISC.

Name	FLIP.
Address	14544
Rom #	0
Romjsb	N

Run time code for the BASIC statement:

FLIP

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

DR	AR	DC	E	ST	PTR1	PTR2
36	-	-	-	U	-	-

The actual code for FLIP is:

```
FLIP.    LDB R36,=200
         STBD R36,=KEYSTS
         RTN
```

FNDLIN
MISC.

Name	FNDLIN
Address	32355
Rom #	0
Romjsb	N

Finds a specified line (by number) in a BASIC program.

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

DR	AR	DC	E	ST	PTR1	PTR2
65	6	B	*	U	*	-

INPUT REGISTER CONTENTS

R75-R77 = Line number to be found

OUTPUT CONDITIONS

E = 0 if the line was found.
E = 17 if the line was not found.
PTR1 points to the line length byte of the desired line, if found, else it points to the same byte of the next highest line.

NOTE: Upon return a LDBI R20,=PTR1- would load into R20 the length of the found line, that is PTR1 is really pointing to the least significant byte of the line number. All registers are saved and restored.

FORMAR — PARSE

Name	FORMAR
Address	27034
Rom #	0 **Romjsb** Y

Parses an array variable reference with no subscripts, as in

PRINT# 1; A(,)

SCAN must have been done at entry.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	+

INPUT CONDITIONS

```
R10-R11 = Pointer to input stream
R14     = Current token
R20     = Next character

PTR2    = Pointer to output stream
```

OUTPUT CONDITIONS

```
E=1 if successful
E=0 if unsuccessful
```

FP5 — MATH

Name	FP5
Address	54665
Rom #	0 **Romjsb** N

Runtime code for the system function FP.

Returns the fractional portion of X.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
          X-value    (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
          FP(X) result  (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

```
R40-R47 = Copy of result
```

FRAME. — CRT

Name	FRAME.
Address	66165
Rom #	1 **Romjsb** Y

Runtime code for the BASIC statement

FRAME

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

G$N
PARSE

Name	G$N		
Address	24543		
Rom #	0	Romjsb	Y

Parses one string and one numeric parameter and pushes the incoming token.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS
   R10-R11 = Pointer to input stream
   R14     = Current token
   R20     = Next character

   PTR2    = Pointer to output stream

OUTPUT CONDITIONS
   R10-R11 = Pointer to input stream
   R14     = Next token
   R20     = Next character
   R40-R47 = Set by SCAN

OUTPUT EMC POINTER

      PTR2----> R14 at entry     (1 byte)
                Numeric parameter tokens
                                (x bytes)
                String parameter tokens
                                (x bytes)
```

G$N+NN
PARSE

Name	G$N+NN		
Address	24642		
Rom #	0	Romjsb	Y

Parses one string and 3 numeric parameters and pushes the incoming token.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS
   R10-R11 = Pointer to input stream
   R14     = Current token
   R20     = Next character

   PTR2    = Pointer to output stream

OUTPUT CONDITIONS
   R10-R11 = Pointer to input stream
   R14     = Next token
   R20     = Next character
   R40-R47 = Set by SCAN

OUTPUT EMC POINTER

      PTR2----> R14 at entry     (1 byte)
                Numeric parameter tokens
                                (x bytes)
                String parameter tokens
                                (x bytes)
```

G/A
CRT

Name	G/A		
Address	11606		
Rom #	0	Romjsb	N

Toggles ALPHA to GRAPH and GRAPH to ALPHA (same as the A/G key).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

GØ12N PARSE

Name	GØ12N
Address	24707
Rom #	0
Romjsb	Y

Parses 0, 1, or 2 line number references and pushes the incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	*

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT EMC POINTER

  PTR2----> Incoming token (1 byte)
            x
            x   3 byte line #
            x
            32  Integer constant token
            x
            x   3 byte line #
            x
            32  Integer constant token
NOTE:
  Either or both line numbers are optional
 so the output stream may be different.
```

GØ1N PARSE

Name	GØ1N
Address	24726
Rom #	0
Romjsb	Y

Parses zero or one line number reference and pushes the incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	*

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT EMC POINTER

  PTR2----> Incoming token (1 byte)
            x
            x   3 byte line #
            x
            32  Integer constant token

NOTE: The line number is optional so the
output stream may be different.
```

GØOR2N PARSE

Name	GØOR2N
Address	24744
Rom #	0
Romjsb	Y

Parses zero or two numeric parameters and then pushes the incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R34     = Number of parameters found

OUTPUT EMC POINTER

     PTR2----> R14 at entry     (1 byte)
               Numeric parameter tokens
                                (x bytes)
```

G12OR4
PARSE

Name	G12OR4
Address	24772
Rom #	0
Romjsb	Y

Parses one, two, or four line number references and pushes the incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	U	*

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R34     = Number of parameters found

OUTPUT EMC POINTER

    PTR2----> R14 at entry      (1 byte)
              Numeric parameter tokens
                                (x bytes)
```

G1OR2N
PARSE

Name	G1OR2N
Address	24761
Rom #	0
Romjsb	Y

Parses one or two numeric parameters and then pushes the incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R34     = Number of parameters found

OUTPUT EMC POINTER

    PTR2----> R14 at entry      (1 byte)
              Numeric parameter tokens
                                (x bytes)
```

GCHAR
PARSE

Name	GCHAR
Address	21636
Rom #	0
Romjsb	N

Gets the next non-blank character to R20. If the character is a carriage return then the R10 pointer is not incremented.

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

DR	AR	DC	E	ST	PTR1	PTR2
-	-	-	-	-	-	-

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream

OUTPUT CONDITIONS

  R20     = Next non-blank character

The actual code is:

     GCHAR     SAD
               BIN
     GCH1      LDBD R20,R10
               CMB  R20,=15
               JZR  GCHRTN
               POBD R20,+R10
               CMB  R20,=40
               JZR  GCH1
     GCHRTN    PAD
               RTN
```

GCLR.
CRT

Name	GCLR.
Address	62214
Rom #	1
Romjsb	Y

Runtime code for the GCLEAR statement.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

  Top of stack -> Optional Y-ordinal
                  (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->
```

GEQ$.
MATH

Name	GEQ$.
Address	3667
Rom #	0
Romjsb	N

Checks to see if one string is greater than or equal to a second string.

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

DR	AR	DC	E	ST	PTR1	PTR2
70	12	0	U	U	-	-

```
INPUT STACK CONTENTS

        Length of string 'A'    (2-bytes)
        Address of string 'A'   (3-bytes)
        Length of string 'B'    (2-bytes)
        Address of string 'B'   (3-bytes)
  R12---->

OUTPUT STACK CONTENTS

        A>=B True/False value   (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R70-R77 = Copy of true/false value.

 NOTE: The true/false value is =0 if
 false, =1 if true and is in floating-
 point format.
```

GEQ.
MATH

Name	GEQ.
Address	62734
Rom #	0
Romjsb	Y

Tests to see if one number is greater than or equal to a second number.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

```
INPUT STACK CONTENTS

        A-value  (8-bytes)
        B-value  (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        A>=B True/False value  (8-bytes)
  R12---->

  NOTE: The true/false value will always
  be a tagged integer and will be =1 if
  true and =0 if false.
```

GET) PARSE

Name	GET)
Address	23450
Rom #	0 Romjsb Y

Checks R14 for the close parenthesis token (set by SCAN). ERROR is called if not found, else SCANs and returns.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

```
INPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  PTR2    = Pointer to output stream

OUTPUT CONDITIONS

  If successful:
    E=1
    R14     = Next SCAN token
    R20     = Next character
    R40-R47 = Set by SCAN before exit.

  If unsuccessful:
    E=0
    No registers will have been changed
    and ERROR will have been called.
```

GET1N PARSE

Name	GET1N
Address	24557
Rom #	0 Romjsb Y

Parses one numeric parameter and pushes incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R40-R47 = Set by SCAN

OUTPUT EMC POINTER

    PTR2----> R14 at entry      (1 byte)
              Numeric parameter tokens
                                (x bytes)
```

GET2N PARSE

Name	GET2N
Address	24630
Rom #	0 Romjsb Y

Parses two numeric parameters and pushes incoming token (R14).

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

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R40-R47 = Set by SCAN

OUTPUT EMC POINTER

    PTR2----> R14 at entry      (1 byte)
              Numeric parameter tokens
                                (x bytes)
```

GET4N
PARSE

Name	GET4N	
Address	24635	
Rom #	0	Romjsb Y

Parses four numeric parameters and pushes incoming token (R14).

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream

OUTPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R40-R47 = Set by SCAN

OUTPUT EMC POINTER

    PTR2----> R14 at entry      (1 byte)
              Numeric parameter tokens
                               (x bytes)
```

GETCMA
PARSE

Name	GETCMA	
Address	23477	
Rom #	0	Romjsb Y

Checks current token for a comma (at parse time) and if found does a SCAN to get past it, else calls ERROR.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

OUTPUT CONDITIONS

  If comma was found
        R14     = Next token
        R20     = Next character
        R40-R47 = Set by SCAN
        E=1

  If comma was not found ERROR will have
  been called and E=0.
```

GETPA?
PARSE

Name	GETPA?	
Address	24740	
Rom #	0	Romjsb Y

Parses as many numeric parameters as it can and then pushes the incoming token (R14).

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character


  PTR2    = Pointer to output stream

OUTPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R34     = Number of parameters found

OUTPUT EMC POINTER

    PTR2----> R14 at entry      (1 byte)
              Numeric parameter tokens
                               (x bytes)
```

GETPAR
PARSE

Name	GETPAR
Address	24562
Rom #	0 Romjsb Y

Parses as many numeric parameters as it can. If, at entry, R35=0, any number is acceptable. If R35#0, the number parsed must equal that in R35. GETPAR pushes the input token after all of the parameters

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	U	U	U	U	-	U

```
INPUT CONDITIONS
   R10-R11  = Pointer to input stream
   R14      = Current token
   R20      = Next character
   R35    * = Number of parameters to parse.
              (=0 if any number acceptable)
   PTR2     = Pointer to output stream

OUTPUT CONDITIONS
   R10-R11  = Pointer to input stream
   R14      = Next token
   R20      = Next character
   R34      = Number of parameters found

OUTPUT EMC POINTER

     PTR2----> R14 at entry      (1 byte)
               Numeric parameter tokens
                                (x bytes)
```

GLOAD.
CRT

Name	GLOAD.
Address	72510
Rom #	320 Romjsb Y

Runtime code for the BASIC statement

GLOAD <file name>

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

   Top of stack-> File name length (2 bytes)
         File name address (3 bytes)
   R12---->

OUTPUT STACK CONTENTS

           (empty)
   R12---->
```

GOTOSU
PARSE

Name	GOTOSU
Address	30317
Rom #	0 Romjsb Y

Parses a GOTO or GOSUB number or label and SCANs before returning.).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS

   R10-R11 = Pointer to input stream
   R14     = GOTO or GOSUB token
   R20     = Next character

OUTPUT CONDITIONS

   If successful:
       R14     = Next token
       R20     = Next character
       R40-R47 = Set by SCAN
       E#0

   If unsuccessful then E=0
```

GR$. — MATH

Name	GR$.
Address	3614
Rom #	0
Romjsb	N

Checks to see if one string is greater than another string.

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**

DR	AR	DC	E	ST	PTR1	PTR2
70	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        Length of string 'A'    (2-bytes)
        Address of string 'A'   (3-bytes)
        Length of string 'B'    (2-bytes)
        Address of string 'B'   (3-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        A>B True/False value    (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R70-R77 = Copy of true/false value.

NOTE: The true/false value is =0 if false, =1 if true and is in floating-point format.

GR. — MATH

Name	GR.
Address	62705
Rom #	0
Romjsb	Y

Tests to see if one number is greater than a second number.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

INPUT STACK CONTENTS

```
        A-value   (8-bytes)
        B-value   (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        A>B True/False value   (8-bytes)
  R12---->
```

NOTE: The true/false value will always be a tagged integer and will be a 1 if true and a 0 if false.

GRAD. — MATH

Name	GRAD.
Address	62274
Rom #	0
Romjsb	Y

Puts the computer in grads mode for math operations.

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

DR	AR	DC	E	ST	PTR1	PTR2
36	-	-	-	U	-	-

The actual code is:

```
     DEG.      LDB R36,=90C
     STODRG    STBD R#,=DRG
               RTN
               BYT 241
     RAD.      CLB R36
               JMP STODRG
               BYT 241
     GRAD.     CLB R36
               DCB R36
               JMP STODRG
```

GRAFA.
CRT

Name	GRAFA.	
Address	12626	
Rom #	0	Romjsb N

Forces the CRT to GRAPH ALL mode.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

GRAPH
CRT

Name	GRAPH	
Address	12560	
Rom #	0	Romjsb N

If the CRT display is in ALPHA NORMAL mode at entry, it will be switched to GRAPH NORMAL mode, else nothing will be done.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

OUTPUT CONDITIONS

If the CRT is in GRAPH NORMAL mode at entry, it will be in GRAPH NORMAL mode at exit. If the CRT is in GRAPH ALL mode at entry, it will be in GRAPH ALL mode at exit. If the CRT is in ALPHA NORMAL mode at entry, it will be in GRAPH NORMAL mode at exit. If the CRT is in ALPHA ALL mode at entry, it will be in ALPHA ALL mode at exit. One return address will also be thrown away before returning if it was in ALPHA ALL mode, so it won't return to the calling routine.

GRAPH.
CRT

Name	GRAPH.	
Address	12574	
Rom #	0	Romjsb N

Forces the CRT to GRAPH NORMAL mode.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

GSTOR.
CRT

Name	GSTOR.
Address	72711
Rom #	320 Romjsb Y

Runtime code for the BASIC statement

GSTORE <file name>

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

  Top of stack-> File name length (2 bytes)
         File name address (3 bytes)
  R12---->    .

OUTPUT STACK CONTENTS

         (empty)
  R12---->
```

HLFLIN
CRT

Name	HLFLIN
Address	14110
Rom #	0 Romjsb N

Outputs a string to the CRT, leaving the cursor position at the end of the string.

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**

DR	AR	DC	E	ST	PTR1	PTR2
36	U	B	U	U	-	-

```
INPUT REGISTER CONTENTS

  R26-R27 = Address of the string
  R36-R37 = Length of the string
```

HMCURS
CRT

Name	HMCURS
Address	13661
Rom #	0 Romjsb N

Moves the cursor to the top left of the ALPHA display.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	-	-	-	U	-	-

```
INPUT CONDITIONS

 The cursor must be off (a call to DECUR2
 must have been made) prior to calling
 HMCURS.

OUTPUT CONDITIONS

 CRTBYT will contain the new cursor
 address (which will be the same as the
 contents of CRTRAM.

  R34-R35 = New cursor address.

The actual code for HMCURS is:

HMCURS   LDMD R34,=CRTRAM
         JSB =BYTCRT
         RTN
```

HORN
MISC.

Name	HORN
Address	10400
Rom #	0 **Romjsb** N

Lower-level BEEP entry point.

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

DR	AR	DC	E	ST	PTR1	PTR2
31	U	U	U	U	-	-

```
INPUT CONDITIONS

  R30-R31 = 40,0
  R32-R33 = Frequency as a 4-digit BCD #

  The ARP must be 34 at entry.
  DCM must be BCD at entry.

  R55-R57 = Duration as a 4-digit BCD #

An example of the call sequence:

        LDM R30,=40,0
        LDM R32,=50C,0     ! OR ANY VALUE
        LDM R55,=0,1,0     ! OR ANY VALUE
        BCD
        ARP 34
        JSB =HORN
```

ICOS
MATH

Name	ICOS
Address	77254
Rom #	0 **Romjsb** Y

Returns the inverse cosine of an argument

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	D	U	U	-	-

```
INPUT STACK CONTENTS

        Argument (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

        ACS(Argument) (8 bytes)
  R12---->
```

IDRAW.
CRT

Name	IDRAW.
Address	64706
Rom #	1 **Romjsb** Y

Runtime code for the BASIC statement

IDRAW x,y

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

        X-value    (8 bytes)
        Y-value    (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)    (0 bytes)
  R12---->
```

IMOVE.
CRT

Name	IMOVE.
Address	64643
Rom #	1 Romjsb Y

Runtime code for the BASIC statement

IMOVE x,y

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        X-value    (8 bytes)
       *Y-value    (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)    (0 bytes)
  R12---->
```

INCHR
CRT

Name	INCHR
Address	14262
Rom #	0 Romjsb N

Reads one character from the location in CRT memory pointed to by the CRT's byte address register (CRTBAD).

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

DR	AR	DC	E	ST	PTR1	PTR2
32	-	B	-	U	-	-

INPUT CONDITIONS

The CRT's internal byte address pointer (which is set by storing to CRTBAD) must be pointing to the address of the byte to be read.

OUTPUT REGISTER CONTENTS

R32 = Character from CRT memory.

NOTE: The CPU must be in BIN mode at entry. The actual code for INCHR is:

```
INCHR    DRP 32
         JSB =BUSY     !Refer to CHKSTS
         ICB R#
         STBD R#,=CRTSTS
LOOP2    LDBD R#,=CRTSTS
         JOD LOOP2
         LDBD R32,=CRTDAT
         RTN
```

INF10
MATH

Name	INF10
Address	54321
Rom #	0 Romjsb N

Returns the largest number that can be handled by the computer:

9.99999999999E499

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        (whatever)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (whatever)
        9.99999999999E499   (8-bytes)
  R12---->
```

INIT.
MISC.

Name	INIT.
Address	1241
Rom #	0
Romjsb	Y

Same as the INIT key. Allocates the entire BASIC program.

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

DR	AR	DC	E	ST	PTR1	PTR2
75	6	B	0	U	U	U

INPUT.
MISC.

Name	INPUT.
Address	16314
Rom #	0
Romjsb	N

Does the first half of an INPUT statement (it outputs a '?' and changes CSTAT to 4, idle-in-input).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

INT5
MATH

Name	INT5
Address	54572
Rom #	0
Romjsb	N

Runtime code for the system function FLOOR

Returns the largest integer <= X.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

          X-value (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

          FLOOR(X) result (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R40-R47 = Copy of result
```

INTDIV
MATH

Name	INTDIV
Address	54601
Rom #	0 Romjsb N

Runtime code for the system operater DIV.

Returns the integral portion of the result of A divided by B.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	12	D	U	U	-	-

```
INPUT STACK CONTENTS
           A-value (8-bytes)
           B-value (8-bytes)
   R12----> .

OUTPUT STACK CONTENTS

           A\B result (8-bytes)
   R12---->
```

INTEGR
PARSE

Name	INTEGR
Address	21331
Rom #	0 Romjsb N

Fetches an integer from the input stream at parse-time (such as a line number).

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

DR	AR	DC	E	ST	PTR1	PTR2
22	U	D	#	U	-	-

```
INPUT CONDITIONS

   R10-R11  = Pointer to input stream
   R20      = Next character

OUTPUT CONDITIONS

   R10-R11  = Pointer to input stream
   R20      = Next character (non-digit)
   R22      = Number of digits seen (BCD)
   R36      = Exponent (15C if less than 16
              digits were found)
   R40-R47  = The integer (up to 16 digits)
              The least significant digit
              is in the right nibble of R40.

   E = 0    If some digits were found
   E = 1    If no digits were found
```

INTMUL
MATH

Name	INTMUL
Address	53673
Rom #	0 Romjsb N

Multiplies two 16-bit binary numbers giving a 32-bit binary result.

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

DR	AR	DC	E	ST	PTR1	PTR2
-	-	-	-	-	-	-

```
INPUT REGISTER CONTENTS

   R66-R67 = 16-bit binary number A
   R76-R77 = 16-bit binary number B

OUTPUT REGISTER CONTENTS

   R66-R67 = 16-bit binary number A
   R76-R77 = 16-bit binary number B
   R54-R57 = 32-bit result A*B

 NOTE: INTMUL does a SAD at entry and a
 PAD at exit and saves and restores all
 registers used except for R54-R57.
```

INTORL
MATH

Name	INTORL
Address	57125
Rom #	0
Romjsb	N

Converts numbers from the tagged-integer format to the real (floating-point) format.

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

DR	AR	DC	E	ST	PTR1	PTR2
36	60	D	U	U	-	-

INPUT REGISTER CONTENTS

R60-R67 = Tagged-integer value

OUTPUT REGISTER CONTENTS

R60-R67 = Real value

NOTE: This routine assumes that R60-R67 contains a tagged-integer value. It does not check for a tagged-integer value. Therefore, if you call this routine with a real value in R60-R67 you'll get an indeterminate value returned.

IP5
MATH

Name	IP5
Address	54770
Rom #	0
Romjsb	N

Runtime code for the system function IP.

Returns the integer portions of X.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        X-value (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        IP(X) result (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of result

IPLOT.
CRT

Name	IPLOT.
Address	64660
Rom #	1
Romjsb	Y

Runtime code for the BASIC statement

IPLOT x,y

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        X-value   (8 bytes)
        Y-value   (8 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)   (0 bytes)
R12---->
```

ISIN
MATH

Name	ISIN
Address	77244
Rom #	0 **Romjsb** Y

Returns the inverse sine of an argument.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	D	U	U	-	-

INPUT STACK CONTENTS

```
        Argument (8 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        ASN(Argument) (8 bytes)
R12---->
```

ITAN
MATH

Name	ITAN
Address	77264
Rom #	0 **Romjsb** Y

Returns the inverse tangent of an argument.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	D	U	U	-	-

INPUT STACK CONTENTS

```
        Argument (8 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        ATN(Argument) (8 bytes)
R12---->
```

KEYLA.
MISC.

Name	KEYLA.
Address	13360
Rom #	0 **Romjsb** N

This routine does the same as the KEY LABEL key.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

INPUT CONDITIONS

If R66 = 140 then the CALC mode key labels will be displayed, else the RUN mode key labels will be displayed.

LABEL. CRT

Name	LABEL.
Address	67262
Rom #	1 Romjsb Y

Runtime code for the BASIC statement

LABEL A$

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS
            Length of string  (2 bytes)
            Address of string (3 bytes)
  R12---->

OUTPUT STACK CONTENTS

            (empty)
  R12---->
```

LDIR. CRT

Name	LDIR.
Address	67052
Rom #	1 Romjsb Y

Runtime code for the BASIC statement

LDIR X

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

            LDIR  value (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

            (empty)
  R12---->
```

LEQ$. MATH

Name	LEQ$.
Address	3656
Rom #	0 Romjsb N

Checks to see if one string is less than or equal to a second string.

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

DR	AR	DC	E	ST	PTR1	PTR2
70	12	D	U	U	-	-

```
INPUT STACK CONTENTS

            Length of string 'A'   (2-bytes)
            Address of string 'A'  (3-bytes)
            Length of string 'B'   (2-bytes)
            Address of string 'B'  (3-bytes)
  R12---->

OUTPUT STACK CONTENTS

            A<=B True/False value  (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R70-R77 = Copy of true/false value.

  NOTE: The true/false value is =0 if
  false, =1 if true and is in floating-
  point format.
```

LEQ.
MATH

Name	LEQ.
Address	62662
Rom #	0 Romjsb Y

Tests to see if one number is less than or equal to a second.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

INPUT STACK CONTENTS

```
        A-value (8-bytes)
        B-value (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        A<=B True/False value (8-bytes)
  R12---->
```

NOTE: The true/false value will always be a tagged-integer and will be =1 if true and =0 if false.

LINET.
CRT

Name	LINET.
Address	66336
Rom #	1 Romjsb Y

Runtime code for the BASIC statement

LINE TYPE X

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        Line type value (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

LIST.
PRINT

Name	LIST.
Address	6352
Rom #	0 Romjsb Y

Same as the LIST command.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

This routine will list the BASIC program. It checks the R12 stack for optional list parameters (line numbers) by comparing R12-R13 with top of stack. Be sure they're equal if you don't push any parameters on the stack, or that they're equal before you push one or two parameters. (The parameters would be tagged-integers or floating point numbers.) The listing goes to the CRT IS device.

LN5
MATH

Name	LN5		
Address	52346		
Rom #	0	Romjsb	N

Returns the LN(X).

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
           X value (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
           LN(X) result (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of the result

LOGT5
MATH

Name	LOGT5		
Address	52515		
Rom #	0	Romjsb	N

Returns the LGT(X) (the base 10 logarithm).

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
           X value (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
           LGT(X) result (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of the result

LT$.
MATH

Name	LT$.		
Address	3635		
Rom #	0	Romjsb	N

Checks to see if one string is less than a second string.

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

DR	AR	DC	E	ST	PTR1	PTR2
70	12	D	U	U	-	-

INPUT STACK CONTENTS

```
           Length of string 'A'     (2-bytes)
           Address of string 'A'    (3-bytes)
           Length of string 'B'     (2-bytes)
           Address of string 'B'    (3-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
           A<B True/False value     (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R70-R77 = Copy of true/false value.

NOTE: The true/false value is =0 if false, =1 if true and is in floating-point format.

LT.
MATH

Name	LT.
Address	62643
Rom #	0 **Romjsb** Y

Tests to see if one number is less than a second number.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

INPUT STACK CONTENTS

```
        A-value (8-bytes)
        B-value (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        A<B True/False value (8-bytes)
  R12---->
```

NOTE: The true/false value will always be a tagged-integer and will be =1 if true and =0 if false.

LTCUR.
CRT

Name	LTCUR.
Address	13623
Rom #	0 **Romjsb** N

Moves the cursor left one space on the ALPHA display. (Checks to see if it goes off of the current page of CRT memory and wraps it around if it does.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

LTCURS
CRT

Name	LTCURS
Address	13757
Rom #	0 **Romjsb** N

Moves the cursor address left one space in ALPHA memory. (Doesn't check to see if it goes off of the current page of CRT memory.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

MAX10
MATH

Name	MAX10
Address	56144
Rom #	0 Romjsb N

Returns the larger of two values.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        A-value   (8 bytes)
        B-value   (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        A MAX B  value  (8 bytes)
  R12---->
```

MIN10
MATH

Name	MIN10
Address	56125
Rom #	0 Romjsb N

Returns the smaller of two values.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        A-value   (8 bytes)
        B-value   (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        A MIN B  value  (8 bytes)
  R12---->
```

MOD10
MATH

Name	MOD10
Address	52541
Rom #	0 Romjsb N

Returns X MOD Y

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	D	U	U	-	-

INPUT STACK CONTENTS

```
        X value (8-bytes)
        Y value (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        X MOD Y (8-bytes)
  R12---->
```

MODADR
CRT

Name	MODADR
Address	13255
Rom #	0 **Romjsb** N

Insures that the CRT memory address will remain in the ALPHA memory area when doing address math.

(Especially useful for doing cursor movements.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	76	-	-	U	-	-

INPUT REGISTER CONTENTS

R24-R25 = Displacement of this movement
R34-R35 = New ALPHA memory address

CRTBYT must contain the previous ALPHA memory address. (Thus, R34-R35 will be the contents of CRTBYT plus the contents of R24-R25.)

OUTPUT REGISTER CONTENTS

R34-5 = Address modified for wrap-around

NOTE: The CPU must be in BIN mode before this routine is called.

MODADR checks to see whether the CRT is in ALPHA or ALPHA ALL mode, then checks to see if the new address is past the end of the appropriate boundary. If so, adjusts it to wrap it back to the top of ALPHA memory.

MOVCRS
CRT

Name	MOVCRS
Address	13771
Rom #	0 **Romjsb** N

Moves the cursor address a specified distance.
(Doesn't check to see if it goes off of the current page of CRT memory.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

R24-R25 = Offset from current cursor location to desired new location.

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

MOVDN
MISC.

Name	MOVDN
Address	57172
Rom #	0 **Romjsb** N

Moves a block of memory from lower addresses to higher addresses, starting with the highest address and working DOWN to the lowest. (Works in lower 64K address range only.)

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

DR	AR	DC	E	ST	PTR1	PTR2
50	6	-	-	U	-	-

INPUT REGISTER CONTENTS

Assumes BIN mode at entry.
R22-R23 = Number of bytes to be moved.
R24-R25 = Pointer to the first word of source block to be moved (the highest addressed byte).
R26-R27 = Pointer to the first word of the sink block to be moved into (the highest addressed byte).

OUTPUT REGISTER CONTENTS

R22-R23 = 0
R24-R25 = Pointer to the last word of the source block to be moved (the lowest addressed byte).
R26-R27 = Pointer to the last word of the sink block to be moved into (the lowest addressed byte).

NOTE: For moves involving extended memory, use the routine EMOVUP.

MOVE. CRT

Name	MOVE.
Address	64634
Rom #	1
Romjsb	Y

Runtime code for the BASIC statement

MOVE x,y

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

           X-value (8 bytes)
           Y-value (8 bytes)
   R12---->

OUTPUT STACK CONTENTS

           (empty)
   R12---->
```

MOVUP MISC.

Name	MOVUP
Address	57232
Rom #	0
Romjsb	N

Moves a block of memory from higher addresses to lower addresses, starting with the lowest address and working UP to the highest. (Works in lower 64K address range only.)

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

DR	AR	DC	E	ST	PTR1	PTR2
50	6	-	-	U	-	-

```
INPUT REGISTER CONTENTS

   Assumes BIN mode at entry.
   R22-R23 = Number of bytes to be moved.
   R24-R25 = Pointer to the first word of
         source block to be moved (the
         lowest  addressed byte).
   R26-R27 = Pointer to the first word of
         the sink block to be moved into
         the lowest addressed byte.

OUTPUT REGISTER CONTENTS

   R22-R23 = 0
   R24-R25 = Pointer to the last word of
         the source block to be moved (the
         highest addressed byte).
   R26-R27 = Pointer to the last word of
         the sink block to be moved into
         the highest addressed byte.

   NOTE: For moves involving extended
   memory, use the routine EMOVDN.
```

MPY10 MATH

Name	MPY10
Address	53357
Rom #	0
Romjsb	N

Multiplies two real numbers.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS
           (empty)
   R12---->

INPUT REGISTER CONTENTS
   R40-R47 = Real number A (8-bytes)
   R50-R57 = Real number B (8-bytes)

OUTPUT STACK CONTENTS
           Result   A*B (8-bytes)
   R12---->

OUTPUT REGISTER CONTENTS
   R40-R47 = Copy of result   A*B

   NOTE: The CPU must be in BCD mode
   before calling this routine.  The two
   arguments must be real values or the
   result will be unknown.
```

MPYROI
MATH

Name	MPYROI
Address	53517
Rom #	0 Romjsb N

Multiplies one real or tagged-integer number with a second real or tagged-integer number. (This is the main runtime entry point for the system operator *.)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
        Real or tagged-integer A (8-bytes)
      * Real or tagged-integer B (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        Result   A*B (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

```
  R40-R47 = Copy of result  A*B
```

NOTE: The CPU must be in BCD mode before calling this routine.

MSCRE.
DISC

Name	MSCRE.
Address	65176
Rom #	320 Romjsb Y

Runtime code for the BASIC statement

CREATE A$,X,Y

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

INPUT STACK CONTENTS

```
  Top of stack-> File name length (2 bytes)
                 File name address (3 bytes)
                 Number of records (8 bytes)
                 Optional number of bytes/record
                                      (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

MSPRNT
DISC

Name	MSPRNT
Address	66221
Rom #	320 Romjsb Y

Sets the file print pointers to the appropriate file buffer. Part of the statement

PRINT# 1

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

INPUT STACK CONTENTS

```
  Top of stack-> Buffer number (8 bytes)
                 Optional record # (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

NOTE: For figuring out what routines to call and in which order when reading from or printing to disc data files, first write it as a BASIC statement first line of a program. Using the MEM command, look into memory (the line will be at FWCURR-40) to see what the token form is. Refer to MSPRNU for a list of routines and token numbers.

MSPRNU
DISC

Name	MSPRNU
Address	
Rom #	Romjsb

This is a note which is continued from MSPRNT.

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

DR	AR	DC	E	ST	PTR1	PTR2

The Mass Storage ROM routines and their associated token numbes (for PRINT# and READ# statements) are:

```
15    MSPRNT
21    READ.
37    RDNUM.
40    PRARR.
41    RDSTR.
42    PRNUM.
43    PREOL.
44    PRSTR.
45    RDARR.
46    PRARR$
47    RDARR$
```

MSPUR.
DISC

Name	MSPUR.
Address	64604
Rom #	320 Romjsb Y

Purges a disc file and optionally all files after that file

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

  Top of stack-> File name length (2 bytes)
                 File name address (3 bytes)
                 Optional value 0  (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

                 (empty)
  R12---->
```

MSREN.
DISC

Name	MSREN.
Address	64724
Rom #	320 Romjsb Y

Renames a file on a disc.

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

  Top of stack-> Old file name len(2 bytes)
                 Old file name address (2 bytes)
                 New file name length (2 bytes)
                 New file name address (3 bytes)
  R12---->

OUTPUT STACK CONTENTS

                 (empty)
  R12---->
```

NARRE+ PARSE

Name	NARRE+
Address	23461
Rom #	0 Romjsb Y

SCANs and parses a simple numeric variable reference as an array reference (that is, MAT A=ZER).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT CONDITIONS

```
R10-R11 = Pointer to input stream
R20     = Next character
PTR2    = Pointer to output stream
```

OUTPUT CONDITIONS

If successful (token found was a 1) then an array reference (token 22) will have been pushed out to PTR2-(output stream) and a SCAN performed. If unsuccessful when ERROR will have been called.

NARREF PARSE

Name	NARREF
Address	23465
Rom #	0 Romjsb Y

This routine parses a simple numeric variable reference as an array reference (that is, MAT A=ZER).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT CONDITIONS

```
R10-R11 = Pointer to input stream
R14     = Next token
R20     = Next character
PTR2    = Pointer to output stream
```

OUTPUT CONDITIONS

If successful (token found was a 1) then an array reference (token 22) will have been pushed out to PTR2-(output stream) and a SCAN performed. If unsuccessful then ERROR will have been called.

NUMCON PARSE

Name	NUMCON
Address	23551
Rom #	0 Romjsb Y

If the next token is a numeric constant it is pushed out to the output stream and SCAN is called.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT CONDITIONS

```
R10-R11 = Pointer to input stream
R14     = Current token
R20     = Next character
```

OUTPUT CONDITIONS

```
If numeric constant was found
    R14     = Next token
    R20     = Next character
    R40-R47 = Set by SCAN
    E=1
```

If numeric constant was not found registers are unchanged and E=0.

NUMVA+
PARSE

Name	NUMVA+
Address	22403
Rom #	0
Romjsb	Y

Calls SCAN, then parses a numeric expression (falls through into NUMVAL).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R20     = Next character
  PTR2    = Pointer to output stream

OUTPUT CONDITIONS

  If successful:
     E#0
     R14      = Next SCAN token
     R20      = Next character
     R40-R47  = Set by SCAN before exit.

  If unsuccessful:
     E=0
     R10 is reset to incoming value so
     other parsing may be tried.
```

NUMVAL
PARSE

Name	NUMVAL
Address	22406
Rom #	0
Romjsb	Y

Parses a numeric expression (any expression that will eventually evaluate down to a single numeric value).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS
  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  PTR2    = Pointer to output stream

OUTPUT CONDITIONS

  If successful:
     E#0
     R14      = Next SCAN token
     R20      = Next character
     R40-R47  = Set by SCAN before exit.

  If unsuccessful:
     E=0
     R10 is reset to incoming value so
     other parsing may be tried.
```

ONEB
MATH

Name	ONEB
Address	12153
Rom #	0
Romjsb	N

Takes one number off of the R12 stack and converts it to a 15-bit signed binary value.

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

DR	AR	DC	E	ST	PTR1	PTR2
76	46	B	U	U	-	-

```
INPUT STACK CONTENTS

        Real or tagged-integer (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

OUTPUT REGISTER CONTENTS

  R46-R47 = 15-bit signed binary number
  R76-R77 = Copy of 15-bit value

  NOTE: If the value is negative then
  R46-R47 will contain the two's
  complement of the absolute value of the
  original argument (that is, the value -1
  would be returned as octal 177777).
```

ONEI
MATH

Name	ONEI
Address	56736
Rom #	0 Romjsb N

Takes one number off of the R12 stack and converts it to the tagged-integer format if it's not already.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

INPUT STACK CONTENTS

```
        Real or tagged-integer (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Tagged-Integer value

ONER
MATH

Name	ONER
Address	56777
Rom #	0 Romjsb N

Takes one number off of the R12 stack and, if it's not in the real (floating point) format, converts it to that format.

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

DR	AR	DC	E	ST	PTR1	PTR2
60	40	D	0	U	-	-

INPUT STACK CONTENTS

```
        Real or tagged-integer (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Real value from R12 stack
R60-R67 = Copy of real value from stack

ONEROI
MATH

Name	ONEROI
Address	57035
Rom #	0 Romjsb N

Takes one number off of the R12 stack and returns a flag to tell whether it is a real or integer format number.

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

DR	AR	DC	E	ST	PTR1	PTR2
44	12	-	U	U	-	-

INPUT STACK CONTENTS

```
        Real or tagged-integer  (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Real or tagged-integer value

```
  E       = 0 if R40-R47 is a real number
          = 1 if R40-R47 is a tagged-integer
```

ONEX
MATH

Name	ONEX
Address	56673
Rom #	0 Romjsb N

Takes one number off of the R12 stack and converts it to a 16-bit unsigned binary value with a separate sign flag.

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

DR	AR	DC	E	ST	PTR1	PTR2
76	46	B	U	U	-	-

INPUT STACK CONTENTS

Real or tagged-integer (8-bytes)
R12---->

OUTPUT STACK CONTENTS

<empty>
R12---->

OUTPUT REGISTER CONTENTS

R46-R47 = 16-bit unsigned binary number
R76-R77 = Copy of 16-bit value
R32 = Sign of 16-bit value
If R32=0 then value is positive
If R32#0 then value is negative

OUTCH1
CRT

Name	OUTCH1
Address	14130
Rom #	0 Romjsb N

Outputs one character to the CRT at the address contained in CRTBYT.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	24	B	-	U	-	-

INPUT CONDITIONS

CRTBYT must contain the address of the CRT memory location the character is to stored into.

R32 = The ASCII code of the character to be output.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

OUTCHR
CRT

Name	OUTCHR
Address	14143
Rom #	0 Romjsb N

Outputs one character to the CRT at the address contained in CRTBYT and scrolls display up if the cursor position moves off of the bottom-right corner.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	-	U	-	-

INPUT CONDITIONS

CRTBYT must contain the address of the CRT memory location the character is to stored into.

R32 = The ASCII code of the character to be output.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

OUTSTR
CRT

Name	OUTSTR
Address	14020
Rom #	0 **Romjsb** N

Outputs a string to the CRT, blanks the rest of the line, moves the cursor to the beginning of the next line and displays the cursor.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

INPUT REGISTER CONTENTS

R26-R27 = Address of the string
R36-R37 = Length of the string

PAGES.
CRT

Name	PAGES.
Address	12756
Rom #	0 **Romjsb** N

Sets the CRT to page size 16 or to page size 24 (same as the PAGESIZE statement).

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

INPUT STACK CONTENTS

```
      Page size (16 or 24 decimal)(8 bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
      (empty)
R12---->
```

NOTE: Gives a warning message if the parameter is not equal to 16 or 24.

PAGES1
CRT

Name	PAGES1
Address	13001
Rom #	0 **Romjsb** N

Forces the CRT to page size 16.

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**

DR	AR	DC	E	ST	PTR1	PTR2
36	54	B	U	U	-	-

PAGES2
CRT

Name	PAGES2
Address	13103
Rom #	0
Romjsb	N

Forces the CRT to page size 24.

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**

DR	AR	DC	E	ST	PTR1	PTR2
36	54	B	U	U	-	-

PARSER
PARSE

Name	PARSER
Address	20000
Rom #	0
Romjsb	Y

Parses whatever is in the INPBUF to the internal token form.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	U

If no errors occur, the parsed line will have been edited into the program if it was a program line, else it will be between NXTMEM and SAVPT2.

PI10
MATH

Name	PI10
Address	54374
Rom #	0
Romjsb	N

Returns PI:

3.14159265359

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        <whatever>
  R12---->

OUTPUT STACK CONTENTS

        <whatever>
        PI 3.14159265359     (8-bytes)
  R12---->
```

PLIST.
PRINT

Name	PLIST.
Address	6344
Rom #	0 Romjsb Y

Same as the PLIST command.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

This routine will list the BASIC program. It checks the R12 stack for optional list parameters (line numbers) by comparing R12-R13 with the top of stack. So, be sure they're equal if you don't push any parameters on the stack, or that they're equal before you push one or two parameters. (The parameters would be tagged-integers or floating-point numbers. The listing goes to the PRINTER IS device.

PLOT.
CRT

Name	PLOT.
Address	64652
Rom #	1 Romjsb Y

Runtime code for the BASIC statement

PLOT x,y

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

            X-value (8 bytes)
            Y-value (8 bytes)
   R12---->

OUTPUT STACK CONTENTS

            (empty)
   R12---->
```

POS.
MISC.

Name	POS.
Address	4227
Rom #	0 Romjsb N

Returns a value which is the position in one string of a second string. This is the runtime code for the system function POS.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

```
INPUT STACK CONTENTS

          Length of arg string 'A'   (2 bytes)
          Address of arg string 'A'  (3 bytes)
          Length of arg string 'B'   (2 bytes)
          Address of arg string 'B'  (3 bytes)
  R12---->

OUTPUT STACK CONTENTS

          Position of string B in string A
  R12---->                           (8 bytes)
```

NOTE: Position value will be 0 if string B does not exist in string A.

PRARR$
DISC

Name	PRARR$
Address	70730
Rom #	320 Romjsb Y

Prints an entire string array to a data file buffer.

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Abs. address of first element of
           the array       (3 bytes)
        Abs. address of the name of the
           variable        (3 bytes)
        Header byte of variable  (1 byte)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->


NOTE: Refer to MSPRNT.
```

PRARR.
DISC

Name	PRARR.
Address	70167
Rom #	320 Romjsb Y

Prints an entire numeric array into a data file buffer.

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Abs. address of first element of
           the array       (3 bytes)
        Abs. address of the name of the
           variable        (3 bytes)
        Header byte of variable  (1 byte)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->


NOTE: Refer to MSPRNT.
```

PRDRVR
PRINT

Name	PRDRVR
Address	73023
Rom #	320 Romjsb Y

This is the printer driver routine. It's similar to the OUTSTR routine, but for an external printer.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

```
INPUT REGISTER CONTENTS

  R26-R27 = Address of first character
            of output buffer (where the
            first character is at the
            lowest address).
  R36-R37 = Number of bytes to be output.
```

PREOL.
DISC

Name	PREOL.
Address	70464
Rom #	320 Romjsb Y

Terminates a print to a data file buffer. This routine must always be called at the end of a series of calls to PRNUM., PRARR., PRSTR., or PRARR$.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	–	U

Refer to MSPRNT.

PRINT.
PRINT

Name	PRINT.
Address	71332
Rom #	0 Romjsb Y

Sets up SCTEMP so that it contains the current PRINTER IS select code. It is usually used prior to calling DRV12.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	U	–	–	U	–	–

PRLINE
PRINT

Name	PRLINE
Address	71641
Rom #	0 Romjsb Y

Dumps either the PRINT buffer or the DISP buffer.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	–	U

DISP. or PRINT. must have been called to set up the select code and buffer pointers before PRLINE was called.

PRNTR.
PRINT

Name	PRNTR.
Address	75631
Rom #	1 Romjsb Y

Runtime code for the PRINTER IS statement.

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**

DR	AR	DC	E	ST	PTR1	PTR2
65	U	U	U	U	-	-

```
INPUT STACK CONTENTS

  TOS-> Select code  (8-bytes)
        Optional line length  (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->
```

PRNUM.
DISC

Name	PRNUM.
Address	67220
Rom #	320 Romjsb Y

Prints a numeric value to a data file buffer.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Value to be printed  (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

NOTE: Refer to MSPRNT.
```

PRSTR.
DISC

Name	PRSTR.
Address	66662
Rom #	320 Romjsb Y

Prints a string to a data file buffer.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Length of string     (2 bytes)
        Address of string    (3 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

NOTE: Refer to MSPRNT.
```

RAD10
MATH

Name	RAD10
Address	54472
Rom #	0 Romjsb N

Runtime code for the system function DTR.

Converts degrees to radians.

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**

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

INPUT STACK CONTENTS

```
        Degree-value   (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        Radians result  (8-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

R40-R47 = Copy of result

RAD.
MATH

Name	RAD.
Address	62267
Rom #	0 Romjsb Y

Puts the computer in radians mode for math operations.

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

DR	AR	DC	E	ST	PTR1	PTR2
36	-	-	-	U	-	-

The actual code is:

```
    DEG.      LDB R36,=90C
    STODRG    STBD R#,=DRG
              RTN
              BYT 241
    RAD.      CLB R36
              JMP STODRG
              BYT 241
    GRAD.     CLB R36
              DCB R36
              JMP STODRG
```

RDARR$
DISC

Name	RDARR$
Address	70312
Rom #	320 Romjsb Y

Reads an entire string array from a data file buffer into a string array variable area.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

INPUT STACK CONTENTS

```
        Abs. address of first element of
            the array        (3 bytes)
        Abs. address of the name of the
            variable         (3 bytes)
        Header byte of variable   (1 byte)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

NOTE: Refer to MSPRNT.

RDARR. DISC

Name	RDARR.
Address	70106
Rom #	320 Romjsb Y

Reads an entire numeric array from a data file buffer into a numeric array variable area.

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Abs. address of first element of
          the array          (3 bytes)
        Abs. address of the name of the
          variable           (3 bytes)
        Header byte of variable   (1 byte)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

NOTE: Refer to MSPRNT.
```

RDNUM. DISC

Name	RDNUM.
Address	67503
Rom #	320 Romjsb Y

Reads a number from a data file buffer into a variable area.

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Abs. address of variable value
                             (3 bytes)
        Abs. address of the name of the
          variable           (3 bytes)
        Header byte of variable   (1 byte)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

NOTE: Refer to MSPRNT.
```

RDSTR. DISC

Name	RDSTR.
Address	67314
Rom #	320 Romjsb Y

Reads a string from a data file buffer into a string variable.

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

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

```
INPUT STACK CONTENTS

        Abs. address of name (3 bytes)
        Header of variable   (1 byte)
        Max length of string variable area
                             (2 bytes)
        Abs address of first byte of string
                 variable    (3 bytes)
        Max length available to store into
                             (2 bytes)
        Abs address of first byte to store
                 into        (3 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

NOTE: Refer to MSPRNT.
```

READ. — DISC

Name	READ.
Address	66221
Rom #	320 **Romjsb** Y

Sets the file print pointers to the appropriate file buffer. Part of the statement

READ# 1

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**

DR	AR	DC	E	ST	PTR1	PTR2
U		U	U	U	-	U

INPUT STACK CONTENTS

```
  Top of stack-> Buffer number (8 bytes)
     Optional record # (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        (empty)
  R12---->
```

NOTE: Refer to MSPRNT.

REFNUM — PARSE

Name	REFNUM
Address	27530
Rom #	0 **Romjsb** Y

Parses a simple numeric or array variable as a store variable token. A SCAN must have been done before calling. (This routine changes fetch variable tokens 1 and 2 into store tokens 21 and 22.)

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	#

INPUT CONDITIONS

```
  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

  PTR2    = Pointer to output stream
```

OUTPUT CONDITIONS

```
  E#0 if successful
  E=0 if unsuccessful
```

RELMEM — MISC.

Name	RELMEM
Address	31777
Rom #	0 **Romjsb** N

This routine will release all temporary memory that was reserved by calling RESMEM.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

NOTE: The system uses the RAM location known as RMEM to keep track of the amount of memory currently reserved.

REM10
MATH

Name	REM10
Address	52533
Rom #	0 **Romjsb** N

Returns the remainder
RMD(X,Y)

=X-Y*IP(X/Y)

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

DR	AR	DC	E	ST	PTR1	PTR2
U	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        X value    (8-bytes)
        Y value    (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        Remainder  (8-bytes)
  R12---->
```

RESET.
MISC.

Name	RESET.
Address	5407
Rom #	0 **Romjsb** Y

Does the same as the RESET key.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	U	U

Call this routine with a 'JSB =' or a 'JSB =ROMJSB' instruction, the same as any other routine. (It doesn't mess up the R6 stack.)

Refer to the Owner's Manual to find out what RESET does to the computer memory and status.

RESMEM
MISC.

Name	RESMEM
Address	31741
Rom #	0 **Romjsb** N

Reserves temporary scratchpad memory. (It gets released at the end of each line of a BASIC program and at each @ sign (concatenation of statements).)

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	B	U	U	-	-

```
INPUT REGISTER CONTENTS

  R55-R57 = Number of bytes to be reserved.

OUTPUT REGISTER CONTENTS

  R55-R57 = Number of bytes reserved (same
                as input)
  R65-R67 = Address of highest byte + 1

NOTE:
     E=0 if memory was reserved OK.
     E#0 if there was an error (MEM OVF).
```

The address that is returned in R65 is such that the following code will store a byte into the highest addressed location of the block reserved:

```
STMD R65,=PTR2
STBI R36,=PTR2-
```

RETRHI
CRT

Name	RETRHI
Address	13234
Rom #	0
Romjsb	N

Waits until the CRT controller is in a retrace period.

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

DR	AR	DC	E	ST	PTR1	PTR2
31	-	-	-	U	-	-

The actual code for this routine is:

```
RETRHI   DRP R31
DISPLY   LDBD R#,=CRTSTS   !GET CRT STATUS
         ANM R#,=20        !GET RETRACE BIT
         JZR DISPLY        !JIF DISP TIME
         RTN               !ELSE RETURN
```

This routine would be used when manually switching CRT modes (ALPHA/GRAPHICS, BLANKED/UNBLANKED, etc.). If you switch in the middle of a display period, you may get an ugly flash.

RND10
MATH

Name	RND10
Address	53741
Rom #	0
Romjsb	N

Returns the next pseudo-random number (a value between 0 and 1).

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

          (whatever)
  R12---->

OUTPUT STACK CONTENTS

          (whatever)
          Pseudo-random number (8-bytes)
  R12---->
```

RNDIZ.
MATH

Name	RNDIZ.
Address	55115
Rom #	0
Romjsb	Y

Runtime code for the RANDOMIZE statement.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

```
INPUT STACK CONTENTS

  Top of stack-> Optional RANDOMIZE value
                 (8 bytes)
  R12---->
OUTPUT STACK CONTENTS

          (empty)
  R12---->
```

NOTE: If no parameter is passed to this routine then the contents of R12 and the contents of the top of stack must be equal. If a parameter is passed then R12 must have been stored into the top of stack before the parameter was pushed onto the stack. In other words, the top of stack must be pointing to the first byte of the parameter.

ROMINI
MISC.

Name	ROMINI
Address	6055
Rom #	0
Romjsb	Y

Calls the INIT routines in all of the bank-selectable ROMs and all of the binary programs.

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

DR	AR	DC	E	ST	PTR1	PTR2
0	U	U	U	U	U	U

```
INPUT CONDITIONS

  ROMFL = Reason for the call:

          0 Power on
          1 Reset
          2 Scratch
          3 Loadbin
          4 Run,Init
          5 Load
          6 Stop,Pause
          7 Chain
         10 Allocate class >56
         11 De-allocate class >56
         12 De-compile class >56
         13 Program halt on error

NOTE: ROMINI falls through into BPINI.
Binary programs must insure that R0 does
not get destroyed during their INIT
routine as R0 is used as a counter by
BPINI.
```

ROMJSB
MISC.

Name	ROMJSB
Address	6223
Rom #	0
Romjsb	N

Used for calling a routine in a bank-selectable ROM (address range of 60000 to 77777).

Refer to OUTPUT CONDITIONS for CPU register usage information.

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

DR	AR	DC	E	ST	PTR1	PTR2
*	*	*	*	*	*	*

```
INPUT CONDITIONS
  Calls to ROMJSB must be like this:
            JSB =ROMJSB
            DEF routine name
            BYT rom# of destination routine
  ROMJSB will use the RTN address (on the
  R6 stack from the 'JSB =ROMJSB') to fetch
  the address and rom# you want to call.
  When control returns, it will be to the
  next intstruction after the 'BYT rom#'.
OUTPUT CONDITIONS
  The first four bytes of ERTEMP are
  destroyed by ROMJSB. The DRP=65 and the
  ARP=0 when the destination routine is
  reached. When control returns from
  ROMJSB to your calling routine, DRP,ARP,
  E,status,DCM,and the EMC PTRs are set
  according to the routine that was called.
  R0-R1 are saved on the R6 stack along
  with the number of the ROM that was
  selected when the call was initiated.
  They are restored before ROMJSB returns.
  Other registers are destroyed according
  to the routine that was called.
```

ROMRTN
MISC.

Name	ROMRTN
Address	6207
Rom #	0
Romjsb	N

Reselects ROM 0, then does a RTN. Used by bank-selectable ROMs that need to return to the system, but need to have ROM 0 selected (such as at parse time).

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

DR	AR	DC	E	ST	PTR1	PTR2
0	-	-	-	U	-	-

The actual code for ROMRTN is:

```
ROMRTN    CLB R0
          STBD R0,=RSELEC
          RTN
```

RPLOT.
CRT

Name	RPLOT.
Address	64666
Rom #	1
Romjsb	Y

Runtime code for the BASIC statement

RPLOT x,y

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

        X-value    (8 bytes)
        Y-value    (8 bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->
```

RSTREG
MISC.

Name	RSTREG
Address	22346
Rom #	0
Romjsb	N

Restores some CPU registers from the R6 stack. To be used in conjunction with SAVREG.

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

DR	AR	DC	E	ST	PTR1	PTR2
50	6	-	-	U	-	-

```
INPUT STACK CONTENTS

        (whatever)
        R21-R31
        R30-R37
        R60-R67
  R6---->

OUTPUT STACK CONTENTS

        (whatever)
  R6---->
```

RSUM8K
MISC.

Name	RSUM8K
Address	37670
Rom #	0
Romjsb	N

Used by ROMs perform a checksum on themselves to insure that they haven't gone bad.

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

DR	AR	DC	E	ST	PTR1	PTR2
46	32	-	-	*	-	-

```
INPUT CONDITIONS
   The CPU must be in BIN mode at entry.
   The last two bytes of the ROM
      (addresses 77776 and 77777) must be
      the checksum.
   R32-R33 = Base address of rom (this will
      be 60000 for bank-selectable roms).
OUTPUT CONDITIONS
   Upon exit, the zero flag is set if the
   checksum was good, else it is cleared.
The actual code for RSUM8K is:
   RSUM8K    LDM R34,=377,017  ! 8K/2 + 1
             CLM R40
   RSUM      POMD R36,+R32
             ADM R44,R36
             DCM R34
             JNZ RSUM
             ADM R46,R44
             NCM R46
             CMMD R46,R32
             RTN
```

RTCUR.
CRT

Name	RTCUR.	
Address	13651	
Rom #	0	Romjsb N

Moves the cursor right one space on the ALPHA display. (Checks to see if it goes off of the current page of CRT memory and wraps it around if it does.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

RTCURS
CRT

Name	RTCURS	
Address	13765	
Rom #	0	Romjsb N

Moves the cursor address right one space in ALPHA memory. (Doesn't check to see if it goes off of the current page of CRT memory.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

SAD1
CRT

Name	SAD1	
Address	13723	
Rom #	0	Romjsb N

Changes the start address of the CRT ALPHA display.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	-	-	-	U	-	-

INPUT REGISTER CONTENTS

R34-R35 = New start address

NOTE: The CPU must be in BIN mode before calling SAD1. The start address can only be changed when in ALPHA mode; it is fixed at 10340 (octal) when in GRAPH modes.

The actual code for SAD1 is:

```
SAD1     JSB =RETRHI
         STMD R34,=CRTSAD
         STMD R34,=CRTRAM
         RTN
```

SAVREG
MISC.

Name	SAVREG
Address	22310
Rom #	0
Romjsb	N

Saves some of the CPU registers on the R6 stack. To be used in conjunction with RSTREG.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	-	*	U	-	-

```
INPUT STACK CONTENTS

        (whatever)
   R6---->

OUTPUT STACK CONTENTS

        (whatever)
        R21-R31
        R30-R37
        R60-R67
   R6---->

E = 0    If no problem
E = 1    If error flagged, stack was full
                              (MEM OVF)
```

SCAN
PARSE

Name	SCAN
Address	21110
Rom #	0
Romjsb	Y

Gets the next token from the input stream

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

DR	AR	DC	E	ST	PTR1	PTR2
14	36	B	0	U	-	-

```
INPUT CONDITIONS
   R20       = Next char. from input stream
   R10-R11   = Pointer to input stream
OUTPUT CONDITIONS
   R10-R11   = Pointer to input stream
   R14       = Next token
   R20       = Next character
   R40       = First character searched
   R41-R42   = ROM# (if R42=0)
               or binary program base address
                     (if R42#0)
   R43       = ROM or binary program token #
               or Type if variable
   R44-R46   = If variable, R44-R45 = pointer
                     to name and R46 = length
                     of name
               or integer value
               or secondary attributes for
                                 functions
   R47       = Class (primary attribute)
```

SCAN+
PARSE

Name	SCAN+
Address	21105
Rom #	0
Romjsb	Y

Same as SCAN except it does a JSB =GCHAR first.

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

DR	AR	DC	E	ST	PTR1	PTR2
14	36	B	0	U	-	-

```
INPUT CONDITIONS

   R10-R11   = Pointer to input stream
OUTPUT CONDITIONS
   R10-R11   = Pointer to input stream
   R14       = Next token
   R20       = Next character
   R40       = First character searched
   R41-R42   = ROM# (if R42=0)
               or binary program base address
                     (if R42#0)
   R43       = ROM or binary program token #
               or Type if variable
   R44-R46   = If variable, R44-R45 = pointer
                     to name and R46 = length
                     of name
               or integer value
               or secondary attributes for
                                 functions
   R47       = Class (primary attribute)
```

SCRAT.
MISC.

Name	SCRAT.
Address	5601
Rom #	0 Romjsb Y

Scratches memory, the same as the BASIC command SCRATCH.

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

DR	AR	DC	E	ST	PTR1	PTR2
45	U	B	U	U	U	U

SCRDN
CRT

Name	SCRDN
Address	13671
Rom #	0 Romjsb N

Scrolls the ALPHA display down one line.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	6	B	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

OUTPUT REGISTER CONTENTS

R34-R35 = New start address.

SCRUP
CRT

Name	SCRUP
Address	13736
Rom #	0 Romjsb N

Scrolls the ALPHA display up one line.

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

DR	AR	DC	E	ST	PTR1	PTR2
34	6	B	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

OUTPUT REGISTER CONTENTS

R34-R35 = New start address.

SEC1Ø
MATH

Name	SEC10
Address	54260
Rom #	0 Romjsb N

Returns SEC(X)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT STACK CONTENTS

```
          X value      (8-bytes)
  R12----> 
```

OUTPUT STACK CONTENTS

```
          SEC(X) value    (8-bytes)
  R12---->
```

SEMIC$
PRINT

Name	SEMIC$
Address	72155
Rom #	0 Romjsb Y

Prints a string to the print or display buffer. Same as:

PRINT "ABC";

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
          Length of string     (2 bytes)
          Address of string    (3 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
          (empty)
  R12---->
```

NOTE: DISP. or PRINT. must be called prior to calling SEMIC$ to set up the select code and buffer pointers.

SEMIC.
PRINT

Name	SEMIC.
Address	72274
Rom #	0 Romjsb Y

Prints a number to the print or display buffer. Same as:

PRINT 34;

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
          Number to be printed (8 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
          (empty)
  R12---->
```

NOTE: DISP. or PRINT. must be called prior to calling SEMIC. to set up the select code and buffer pointers.

SEQNO
PARSE

Name	SEQNO
Address	30426
Rom #	0 Romjsb Y

SCANs and tries to parse a line number.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream

  R20     , = Next character

OUTPUT CONDITIONS

  If successful:
       R14      = Next token
       R20      = Next character
       R40-R47  = Set by SCAN
       E#0

  If unsuccessful then E=0
```

SEQNO+
PARSE

Name	SEQNO+
Address	30422
Rom #	0 Romjsb Y

Pushes out the incoming token and then SCANs and parses a line number reference.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Current token
  R20     = Next character

OUTPUT CONDITIONS

  If successful:
       R14      = Next token
       R20      = Next character
       R40-R47  = Set by SCAN
       E#0

  If unsuccessful then E=0
```

SET240
MISC.

Name	SET240
Address	21071
Rom #	0 Romjsb N

Sets immediate break bits (5 and 7) in R17

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

DR	AR	DC	E	ST	PTR1	PTR2
36	6	-	-	U	-	-

The actual code is:

```
SET240   PUBD R36,+R6
         LDB R36,=240
         ORB R17,R36
         POBD R36,-R6
         RTN
```

SGN5
MATH

Name	SGN5		
Address	54202		
Rom #	0	**Romjsb**	N

Returns SGN(X)

(-1 IF X<0, 0 IF X=0, AND +1 IF X>0.)

INPUT STACK CONTENTS

```
        X value     (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        SGN(X) value (8-bytes)
R12---->
```

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

SIN10
MATH

Name	SIN10		
Address	54343		
Rom #	0	**Romjsb**	N

Returns the SIN(X)

INPUT STACK CONTENTS

```
        X value     (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        SIN(X) value    (8-bytes)
R12---->
```

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**

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

SQR5
MATH

Name	SQR5		
Address	53237		
Rom #	0	**Romjsb**	N

Returns the square root of a number:

SQR(X)

INPUT STACK CONTENTS

```
        X value    (8-bytes)
R12---->
```

OUTPUT STACK CONTENTS

```
        SQR(X) value    (8-bytes)
R12---->
```

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	D	U	U	-	-

ST240+
MISC.

Name	ST240+
Address	21067
Rom #	0 Romjsb N

Sets immediate break bits (5 and 7) in R17 and sets R16 to 0 (Idle).

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

DR	AR	DC	E	ST	PTR1	PTR2
36	6	-	-	U	-	-

The actual code is:

```
        ST240+    CLB R16
        SET240    PUBD R36,+R6
                  LDB R36,=240
                  ORB R17,R36
                  POBD R36,-R6
                  RTN
```

STBEEP
MISC.

Name	STBEEP
Address	10441
Rom #	0 Romjsb N

Does a standard BEEP (1.2 kz for 1/10 of a second).

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

DR	AR	DC	E	ST	PTR1	PTR2
31	-	-	-	U	-	-

This is the same BEEP as when an error occurs, or when the BEEP statement is executed with no parameters.

STOST
MISC.

Name	STOST
Address	46472
Rom #	0 Romjsb N

Stores a string into a string variable area.

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**

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
            Abs. address of name          (3 bytes)
            Header of variable            (1 byte)
            Maximum length of variable    (2 bytes)
            Abs. address of first char.   (3 bytes)
            Max length to store into      (2 bytes)
            Abs. address to store into    (3 bytes)
            Len of string to be stored    (2 bytes)
            Abs. address of string to be stored
    R12---->                              (3 bytes)
```

OUTPUT STACK CONTENTS

```
            (empty)
    R12---->
```

NOTE: All but the length and address of the string to be stored will be supplied by the system if you parse the string variable using the parse routine STREEF.

STOSV
MISC.

Name	STOSV		
Address	46057		
Rom #	0	Romjsb	N

Stores a numeric value into a simple numeric or numeric array variable.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT STACK CONTENTS

        Abs. address of variable    (3 bytes)
        Abs. address of name        (3 bytes)
        Header of variable          (1 byte)
        Value to be stored          (8 bytes)
   R12---->

OUTPUT STACK CONTENTS

        (empty)
   R12---->

NOTE: All but the value to be stored will
be supplied by the system if you parse
the variable reference using the parse
routine REFNUM.
```

STRCON
PARSE

Name	STRCON		
Address	24201		
Rom #	0	Romjsb	Y

Parses a quoted string and then calls SCAN.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	*	U	-	U

```
INPUT CONDITIONS

   R10-R11 = Pointer to input stream
   R14     = Next token
   R20     = Next character
   R40-R47 = Set by SCAN
   PTR2    = Pointer to output stream
OUTPUT CONDITIONS

   If successful:
        R14  = Next token
        R20  = Next character
        PTR2 = Pointer to output stream

        E#0

   If unsuccessful:

        E=0
```

STREX+
PARSE

Name	STREX+		
Address	23721		
Rom #	0	Romjsb	Y

SCANs and falls through into STREXP (parses a string expression).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

```
INPUT CONDITIONS

   R10-R11 = Pointer to input stream

   R20     = Next character

OUTPUT CONDITIONS

   If successful:
        R14      = Next token
        R20      = Next character
        R40-R47  = Set by SCAN
        E#0

   If unsuccessful then E=0
```

STREXP
PARSE

Name	STREXP
Address	23724
Rom #	0
Romjsb	Y

Parses a string expression (any expression that will eventually evaluate down to a single string value).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT CONDITIONS

```
R10-R11 = Pointer to input stream
R14     = Current token
R20     = Next character
```

OUTPUT CONDITIONS

```
If successful:
     R14      = Next token
     R20      = Next character
     R40-R47  = Set by SCAN
     E#0

If unsuccessful then E=0
```

STRREF
PARSE

Name	STRREF
Address	24056
Rom #	0
Romjsb	Y

Parses a simple string or string array variable name as a reference (for storing info, as opposed to fetching the value).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	*	U	-	U

INPUT CONDITIONS

```
R10-R11 = Pointer to input stream
R14     = Next token
R20     = Next character
R40-R47 = Set by SCAN
PTR2    = Pointer to output stream
```

OUTPUT CONDITIONS

```
If successful:
     R14  = Next token
     R20  = Next character
     PTR2 = Pointer to output stream

     E#0

If unsuccessful:

     E=0
```

SUB10
MATH

Name	SUB10
Address	52734
Rom #	0
Romjsb	N

Subtracts two real (floating-point) numbers.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

INPUT REGISTER CONTENTS

```
R40-R47 = Real value A (8-bytes)
R50-R57 = Real value B (8-bytes)
```

OUTPUT STACK CONTENTS

```
      Result  B-A (8-bytes)
R12---->
```

OUTPUT REGISTER CONTENTS

```
R40-R47 = Copy of result B-A
```

NOTE: The two numbers must be in floating-point format or the result will be unknown. The CPU must be in BCD mode when when ADD10 is called or the result will be unknown.

SUBROI
MATH

Name	SUBROI	
Address	52724	
Rom #	0	Romjsb N

Subtracts one real or tagged-integer number from a second real or tagged-integer number.
(This is the main runtime entry point for the system operator diadic subtract.)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        Real or tagged-integer A (8-bytes)
        Real or tagged-integer B (8-bytes)
  R12----> .

OUTPUT STACK CONTENTS

        Result A-B (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R40-R47 = Copy of the result
```

NOTE: The result may be either a real or a tagged-integer number. The CPU must be in BCD mode before calling SUBROI.

TAN10
MATH

Name	TAN10	
Address	54363	
Rom #	0	Romjsb N

Returns the TAN(X)

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        X value    (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        TAN(X) value    (8-bytes)
  R12---->
```

TIME.
MISC.

Name	TIME.	
Address	66211	
Rom #	0	Romjsb Y

Runtime code for the system function TIME.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        (whatever)
  R12---->

OUTPUT STACK CONTENTS

        (whatever)
        Time     (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R40-R47 = Copy of time
```

TWOB
MATH

Name	TWOB
Address	56760
Rom #	0 Romjsb N

Takes two numbers off of the R12 stack and converts them to 15-bit signed binary values.

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**

DR	AR	DC	E	ST	PTR1	PTR2
46	26	B	U	U	-	-

```
INPUT STACK CONTENTS
        Real or integer value A (8-bytes)
        Real or integer value B (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

OUTPUT REGISTER CONTENTS
  R26-R27 = 15-bit signed binary number (B)
  R46-R47 = 15-bit signed binary number (B)
  R56-R57 = 15-bit signed binary number (A)
  R76-R77 = 15-bit signed binary number (A)

NOTE: If a value is negative then it will
be represented as the two's complement of
the absolute value of the original
argument (that is, a value of -1 would be
returned as octal 177777).
```

TWOR
MATH

Name	TWOR
Address	57020
Rom #	0 Romjsb N

Takes two numbers off of the R12 stack and if they're in the tagged-integer format they are converted to the floating point (real) format.

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

DR	AR	DC	E	ST	PTR1	PTR2
60	40	D	0	U	-	-

```
INPUT STACK CONTENTS
        Real or tagged-integer A (8-bytes)
        Real or tagged-integer B (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

OUTPUT REGISTER CONTENTS

  R40-7 = Real value (B)
  R50-7 = Real value (A)
  R60-7 = Real value (B)
```

TWORO1
MATH

Name	TWORO1
Address	57050
Rom #	0 Romjsb N

Takes two numbers off of the R12 stack and if both are tagged-integers returns them as such, else does any needed conversion and returns them both as real numbers.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

```
INPUT STACK CONTENTS

        Real or tagged-integer A (8-bytes)
        Real or tagged-integer B (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        (empty)
  R12---->

OUTPUT REGISTER CONTENTS

  R40-7 = Real or tagged-integer (B)
  R50-7 = Real or tagged-integer (A)

  E       = 0 if both numbers are real
          = 1 if both numbers are integers
```

UNEQ$. MATH

Name	UNEQ$.
Address	3603
Rom #	0 Romjsb N

Checks two strings for inequality.

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

DR	AR	DC	E	ST	PTR1	PTR2
70	12	D	U	U	-	-

```
INPUT STACK CONTENTS

        Length of string 'A'    (2-bytes)
        Address of string 'A'   (3-bytes)
        Length of string 'B'    (2-bytes)
        Address of string 'B'   (3-bytes)
  R12---->

OUTPUT STACK CONTENTS

        True/False value (8-bytes)
  R12---->

OUTPUT REGISTER CONTENTS

  R70-R77 = Copy of true/false value.

NOTE: The true/false value is =0 if false,
=1 if true and is in floating-point
format.
```

UNEQ. MATH

Name	UNEQ.
Address	62632
Rom #	0 Romjsb Y

Tests two numbers for inequality.

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

DR	AR	DC	E	ST	PTR1	PTR2
40	12	U	U	U	-	-

```
INPUT STACK CONTENTS

        A-value  (8-bytes)
        B-value  (8-bytes)
  R12---->

OUTPUT STACK CONTENTS

        True/False value  (8-bytes)
  R12---->

NOTE: The true/false value will always be
a tagged-integer and will be =1 if true
and =0 if false.
```

UNQUOT PARSE

Name	UNQUOT
Address	24366
Rom #	0 Romjsb Y

Parses an unquoted string and then calls SCAN. Unquoted strings are terminated by a comma (as in a DATA statement).

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	*	U	-	U

```
INPUT CONDITIONS

  R10-R11 = Pointer to input stream
  R14     = Next token
  R20     = Next character
  R40-R47 = Set by SCAN
  PTR2    = Pointer to output stream
OUTPUT CONDITIONS

  If successful:
      R14  = Next token
      R20  = Next character
      PTR2 = Pointer to output stream

      E#0

  If unsuccessful:

      E=0
```

UPC$. MISC.

Name	UPC$.		
Address	4142		
Rom #	0	**Romjsb**	N

Runtime code for the system function UPC$ and it forces all alpha characters in a string to upper case.

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**

DR	AR	DC	E	ST	PTR1	PTR2
45	6	B	U	U	-	-

INPUT STACK CONTENTS

Length of argument string (2 bytes)
Address of argument string (3 bytes)
R12---->

OUTPUT STACK CONTENTS

Length of result string (2 bytes)
Address of result string (3 bytes)
R12---->

NOTE: The length of the result string will be the same as the length of the argument string but the addresses will be different.

UPCUR. CRT

Name	UPCUR.		
Address	13562		
Rom #	0	**Romjsb**	N

Moves the cursor up one line on the ALPHA display. (Checks to see if it goes off of the current page of CRT memory and wraps it around if it does.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

UPCURS CRT

Name	UPCURS		
Address	13743		
Rom #	0	**Romjsb**	N

Moves the cursor address up one line in ALPHA memory. (Doesn't check to see if it goes off of the current page of CRT memory.)

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**

DR	AR	DC	E	ST	PTR1	PTR2
34	24	-	-	U	-	-

INPUT CONDITIONS

The CPU must be in BIN mode at entry.

CRTBYT must contain the current byte address.

The cursor must be off at entry (a call to DECUR2 will do that).

OUTPUT CONDITIONS

CRTBYT and CRTBAD will be pointing to the new cursor address.

The cursor will still be off.

R34-R35 = The new cursor address.

VAL$.
MISC.

Name	VAL$.	
Address	3770	
Rom #	0	Romjsb Y

Runtime code for the system function VAL$ and it returns the string equivalent of a number.

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

DR	AR	DC	E	ST	PTR1	PTR2
65	12	U	U	U	-	U

INPUT STACK CONTENTS

```
        Numeric argument          (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        Length of string          (2-bytes)
        Address of string         (3-bytes)
  R12---->
```

OUTPUT REGISTER CONTENTS

```
  R65-R67 = Address of string
```

VAL.
MISC.

Name	VAL.	
Address	4020	
Rom #	0	Romjsb Y

Runtime code for the system function VAL and it returns the numeric equivalent of the string argument.

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	U

INPUT STACK CONTENTS

```
        Length of string argument   (2 bytes)
        Address of string argument  (3 bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        Numeric value     (8 bytes)
  R12---->
```

YTX5
MATH

Name	YTX5	
Address	54037	
Rom #	0	Romjsb N

Returns Y^X

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

DR	AR	DC	E	ST	PTR1	PTR2
U	U	U	U	U	-	-

INPUT STACK CONTENTS

```
        Y value    (8-bytes)
        X value    (8-bytes)
  R12---->
```

OUTPUT STACK CONTENTS

```
        Y^X  value  (8-bytes)
  R12---->
```

ZROEXM
MISC.

Name	ZROEXM	
Address	57334	
Rom #	0	Romjsb N

Fills a block of extended memory with blanks (used by the system for initializ-ing string variables)

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

DR	AR	DC	E	ST	PTR1	PTR2
70	6	-	-	U	-	-

INPUT REGISTER CONTENTS

R65-R67 = Number of bytes to filled.
PTR2 = Address of first word +1 of the area to be filled with blanks (the highest address; this routine stores to PTR2-, filling from highest address to lowest).

OUTPUT REGISTER CONTENTS

R65-R67 = 0
PTR2 = Address of first word +1.

NOTE: For filling a block of memory (in the lower 64K address space only) with blanks or zeroes, refer to ZROMEM.

ZROMEM
MISC.

Name	ZROMEM	
Address	44716	
Rom #	0	Romjsb N

Zeroes or blank-fills a block of memory (in the lower 64K address space only).

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**

DR	AR	DC	E	ST	PTR1	PTR2
56	36	-	-	U	-	-

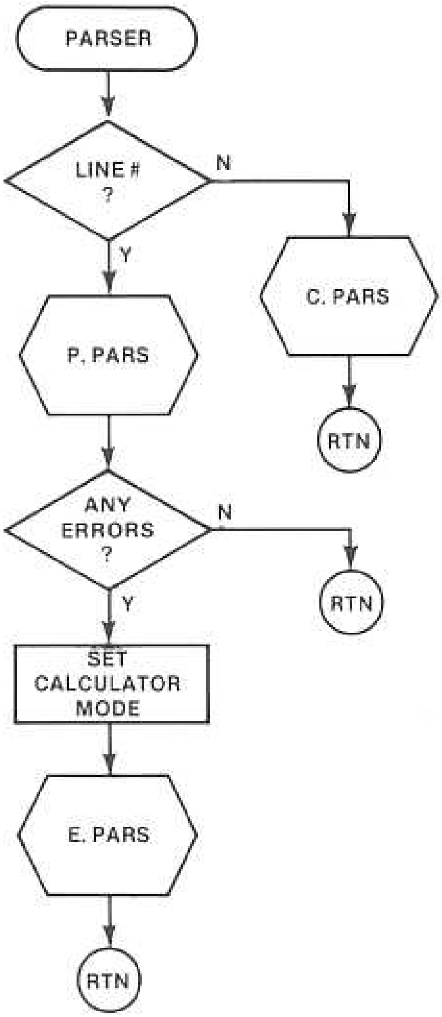
INPUT CONDITIONS

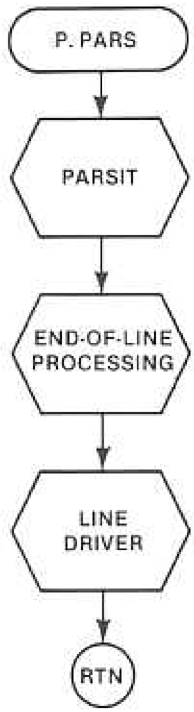
The CPU must be in BIN mode at entry.
If R23=3 then ZROMEM will blank-fill, else it will zero-fill.
R56-R57 = Number of bytes to be filled.
R36-R37 = First byte to be filled (the lowest addressed byte).

NOTE: This routine will only work in the lower 64K address space. There is another routine called ZROEXM that will blank-fill blocks of extended memory but it will not zero-fill.

8.4 Parsing Flow Diagrams

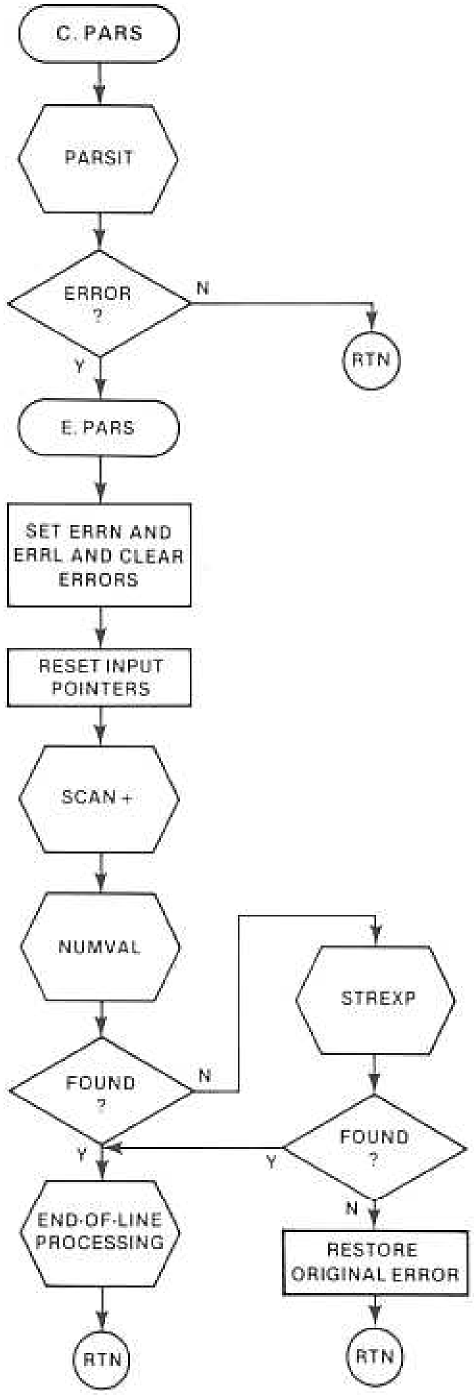
Main Parse Loop



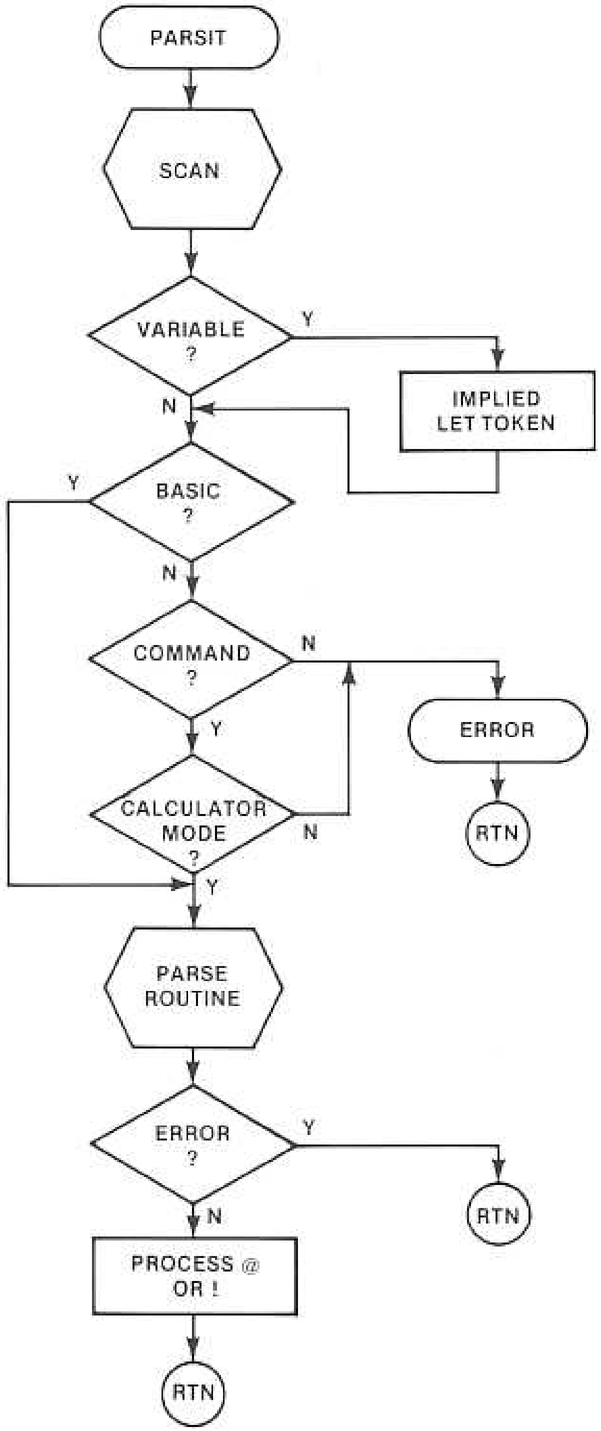


PARSING A PROGRAM LINE

Parsing a Calculator Mode Statement

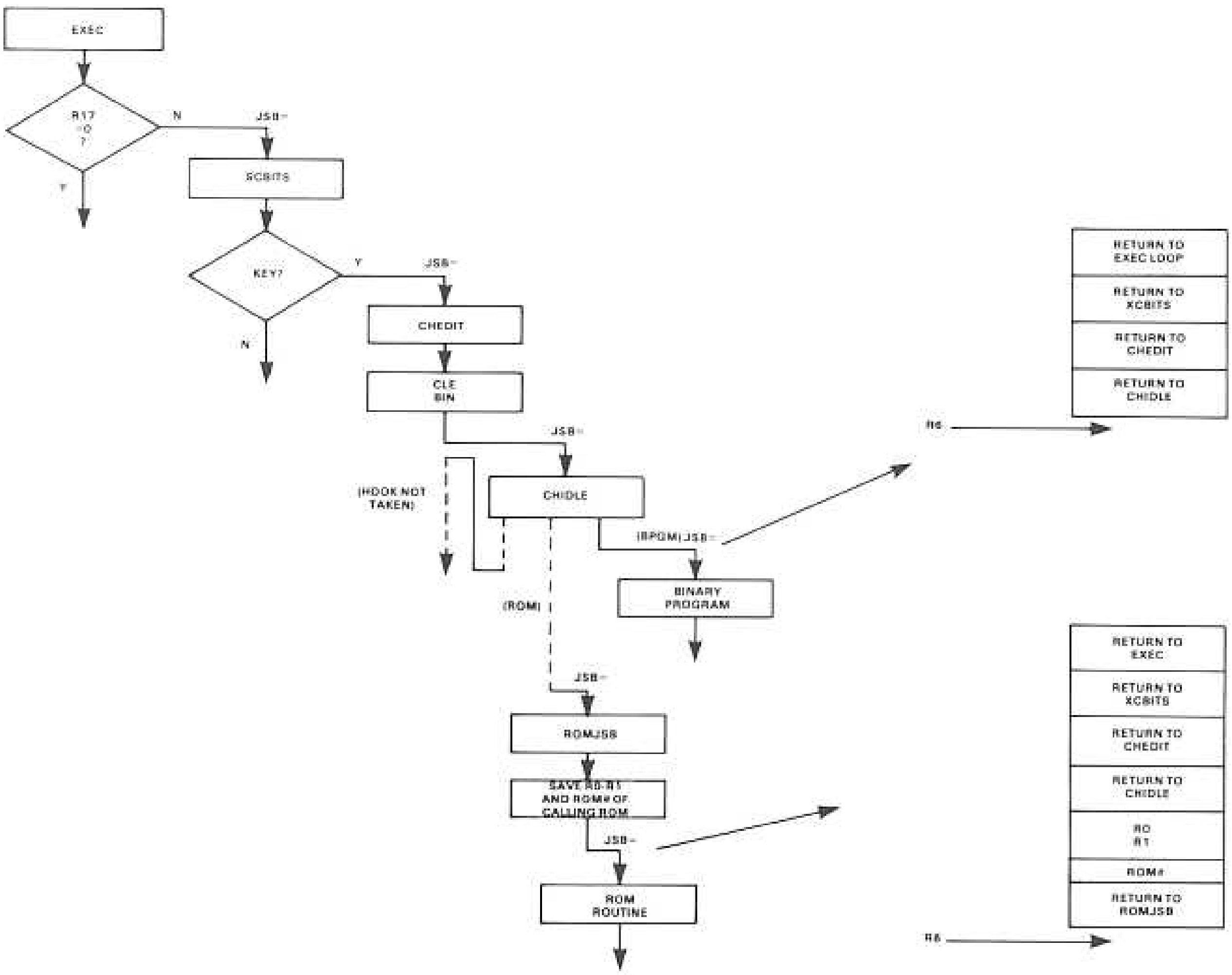


Parsit Routine

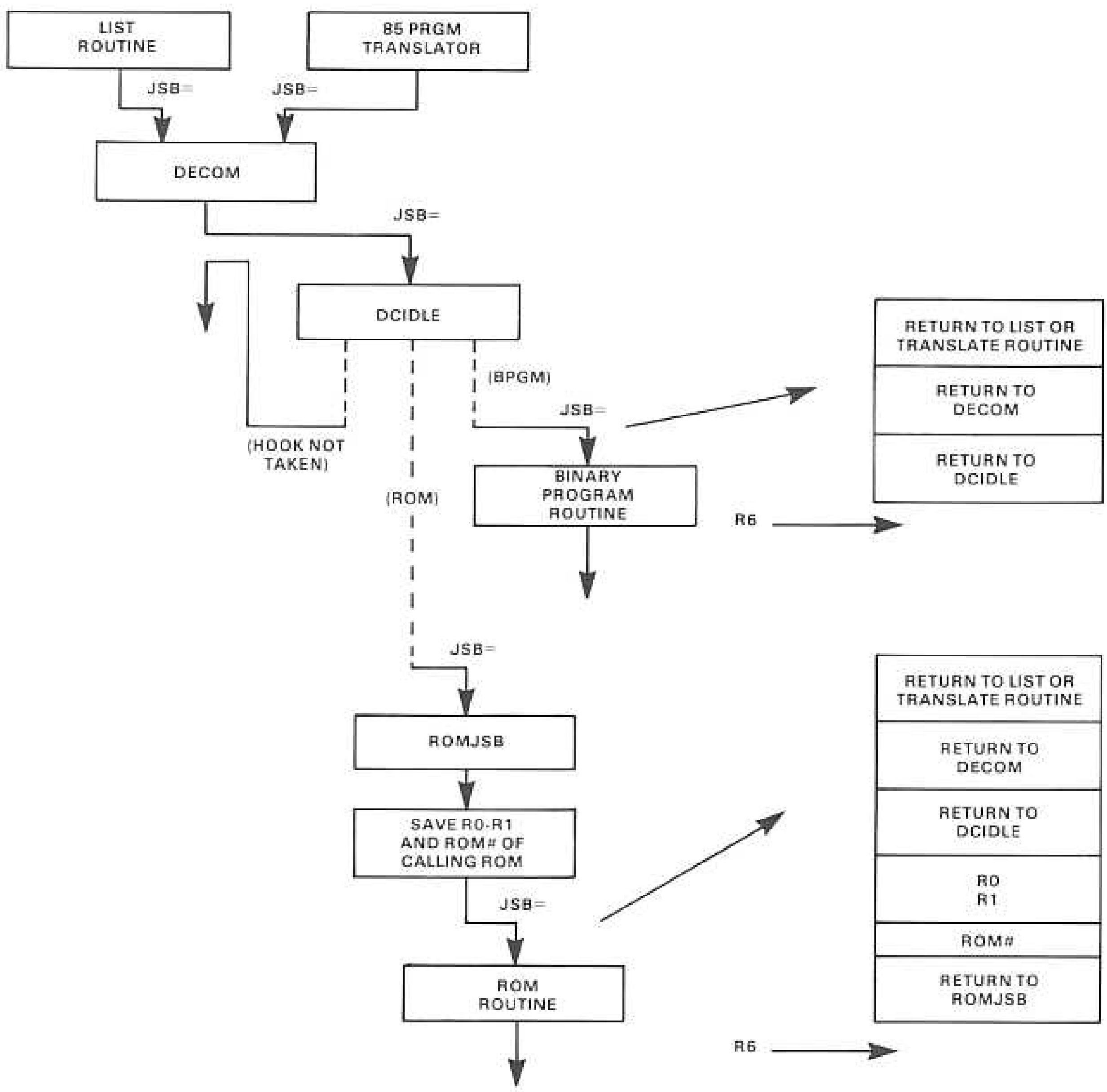


8.5 Hook Flowcharts

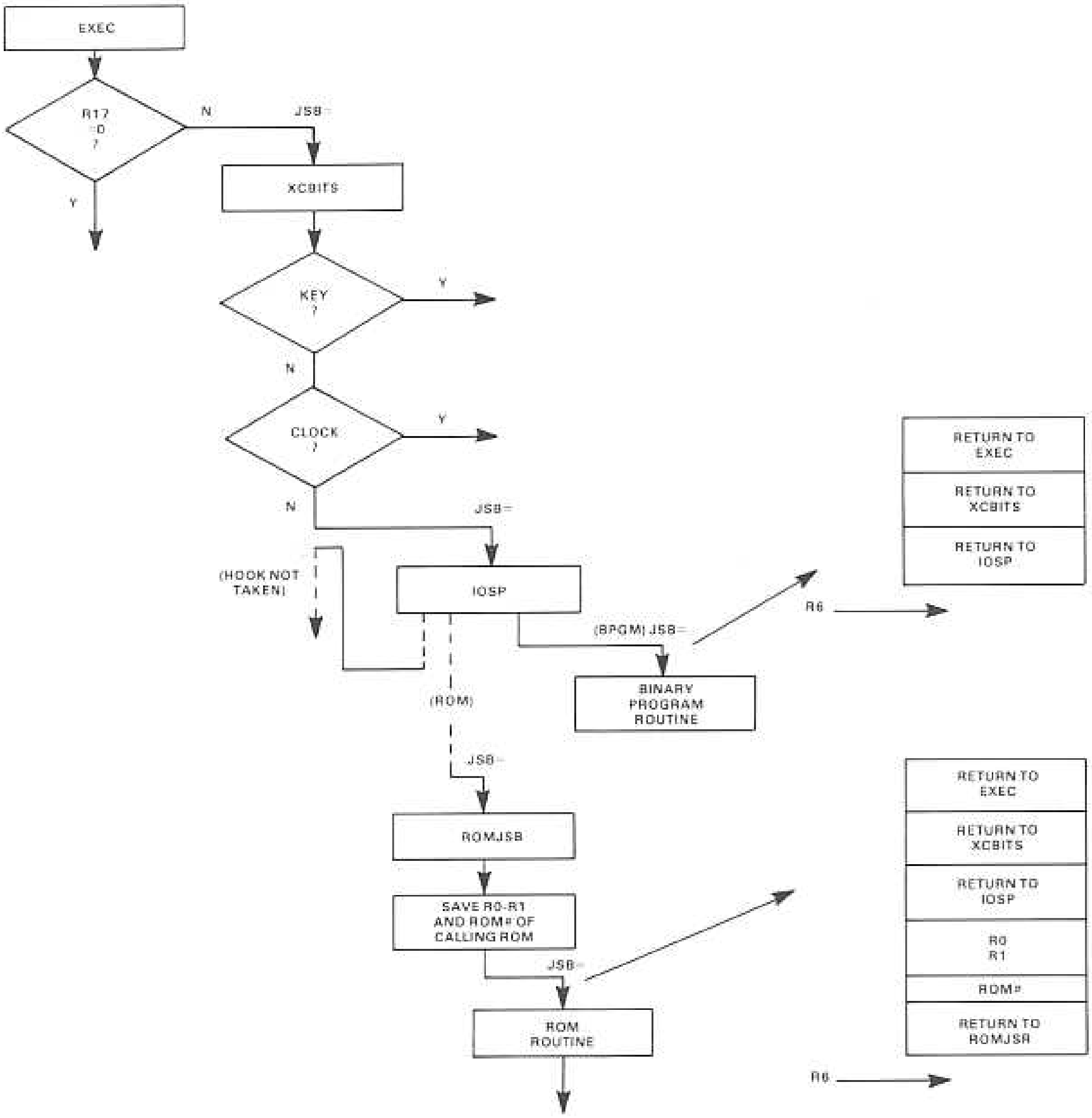
CHIDLE



DCIDLE

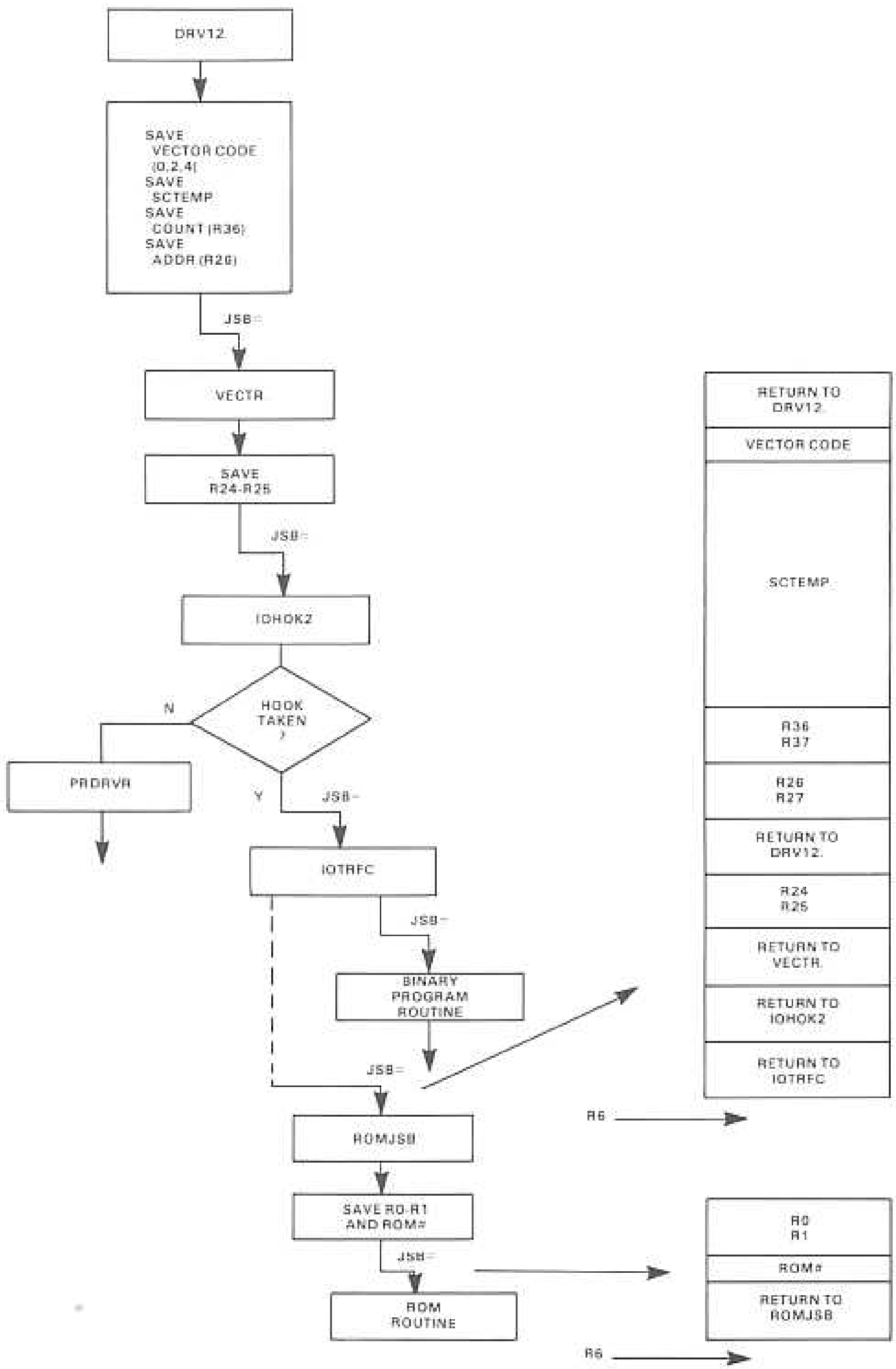

LIST
ROUTINE
85 PRGM
TRANSLATOR
JSB=
JSB=
DECOM
JSB=
DCIDLE
(HOOK NOT
TAKEN)
(BPGM)
JSB=
(ROM)
BINARY
PROGRAM
ROUTINE
RETURN TO LIST OR
TRANSLATE ROUTINE
RETURN TO
DECOM
RETURN TO
DCIDLE
R6
JSB=
ROMJSB
SAVE R0-R1
AND ROM# OF
CALLING ROM
JSB=
ROM
ROUTINE
RETURN TO LIST OR
TRANSLATE ROUTINE
RETURN TO
DECOM
RETURN TO
DCIDLE
R0
R1
ROM#
RETURN TO
ROMJSB
R6

IOSP

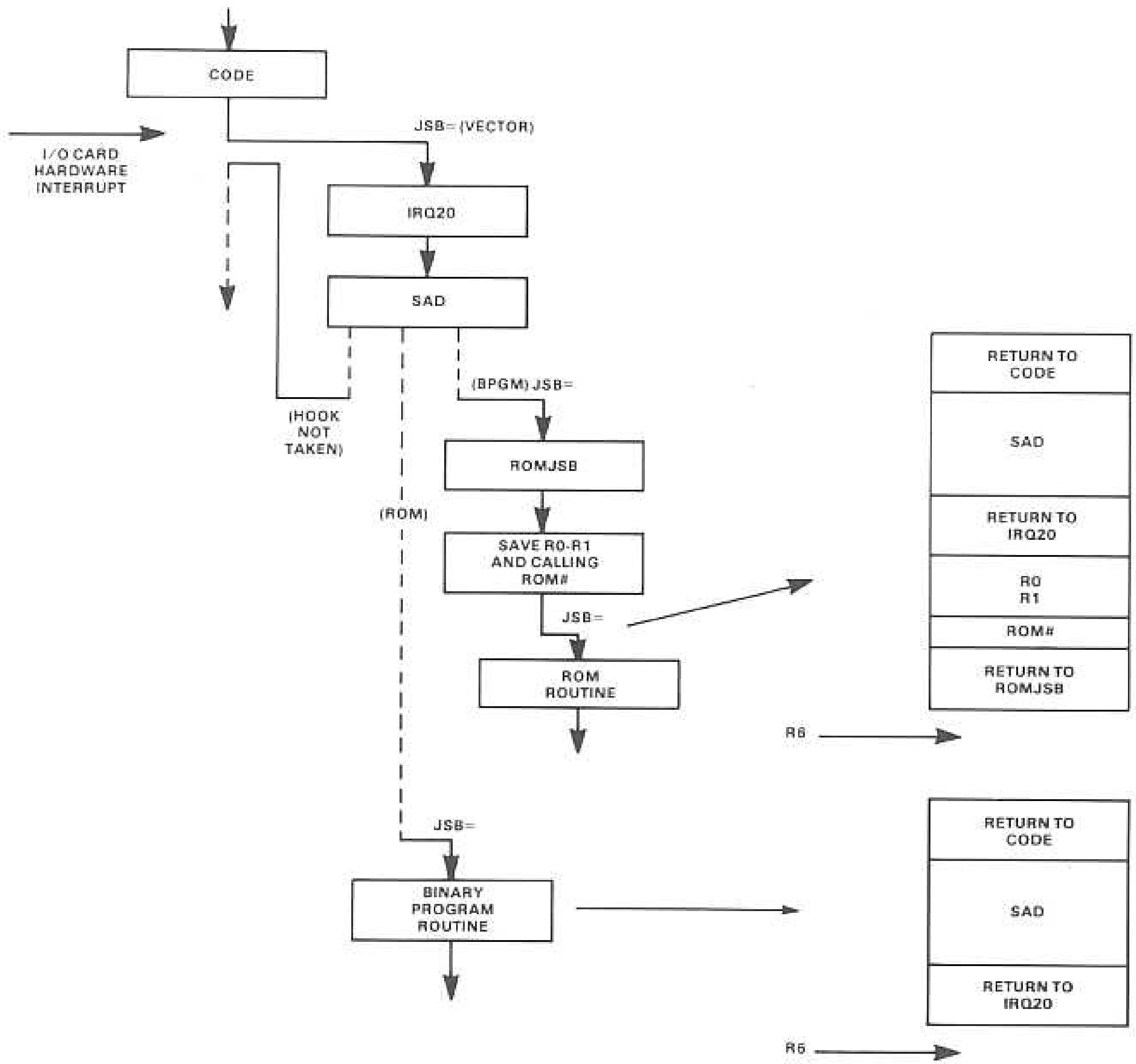


EXEC
R17 =0 ?
N
JSB=
Y
XCBITS
KEY ?
Y
N
CLOCK ?
Y
N
JSB=
(HOOK NOT TAKEN)
IOSP
(BPGM) JSB=
BINARY PROGRAM ROUTINE
(ROM)
JSB=
ROMJSB
SAVE R0-R1 AND ROM# OF CALLING ROM
JSB=
ROM ROUTINE
RETURN TO EXEC
RETURN TO XCBITS
RETURN TO IOSP
R6
RETURN TO EXEC
RETURN TO XCBITS
RETURN TO IOSP
R0
R1
ROM#
RETURN TO ROMJSB
R6

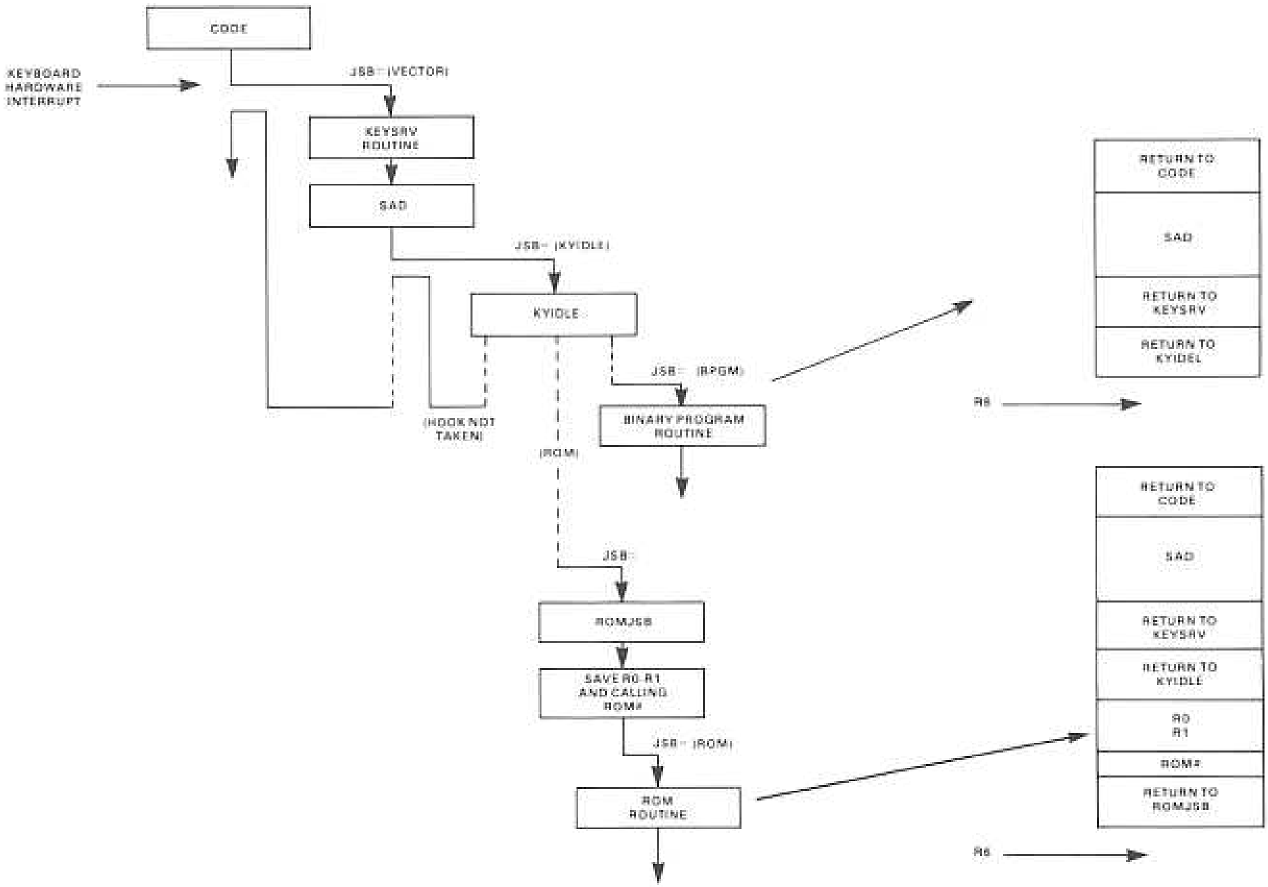
IOTRFC


DRV12
SAVE VECTOR CODE (0,2,4(
SAVE SCTEMP
SAVE COUNT (R36)
SAVE ADDR (R26)
JSB=
VECTR
SAVE R24-R25
JSB=
IOHOK2
HOOK TAKEN ?
N
Y
PRDRVR
JSB=
IOTRFC
JSB=
BINARY PROGRAM ROUTINE
JSB=
ROMJSB
SAVE R0-R1 AND ROM#
JSB=
ROM ROUTINE
RETURN TO DRV12
VECTOR CODE
SCTEMP
R36 R37
R26 R27
RETURN TO DRV12
R24 R25
RETURN TO VECTR
RETURN TO IOHOK2
RETURN TO IOTRFC
R6
R0 R1
ROM#
RETURN TO ROMJSB
R6

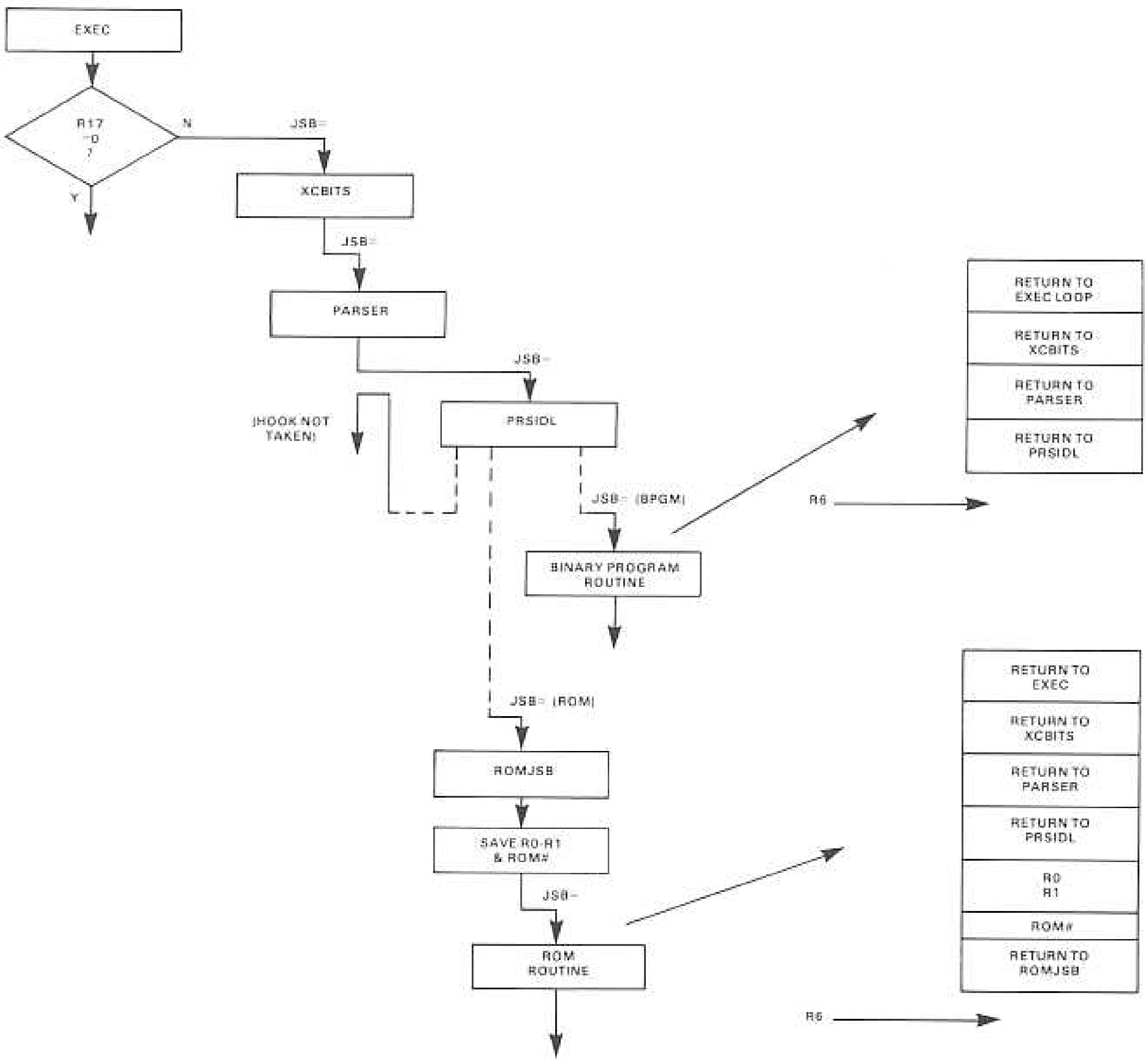
IRQ2Ø


CODE
JSB= (VECTOR)
I/O CARD
HARDWARE
INTERRUPT
IRQ20
SAD
(HOOK
NOT
TAKEN)
(BPGM) JSB=
ROMJSB
(ROM)
SAVE R0-R1
AND CALLING
ROM#
JSB=
ROM
ROUTINE
JSB=
BINARY
PROGRAM
ROUTINE
RETURN TO
CODE
SAD
RETURN TO
IRQ20
R0
R1
ROM#
RETURN TO
ROMJSB
R6
RETURN TO
CODE
SAD
RETURN TO
IRQ20
R6

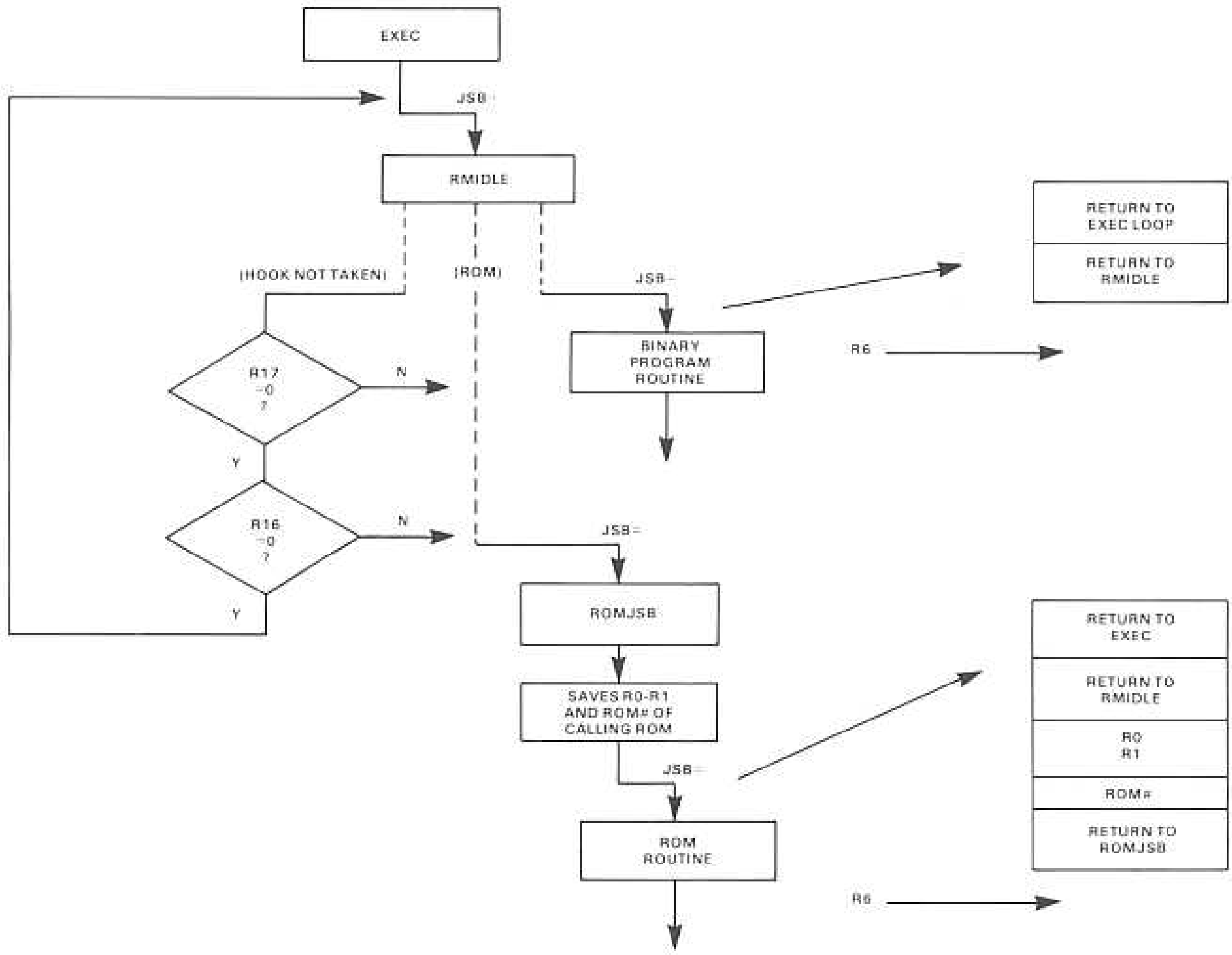
KYIDLE


CODE
KEYBOARD HARDWARE INTERRUPT
JSB = (VECTOR)
KEYSRV ROUTINE
SAD
JSB = (KYIDLE)
KYIDLE
(HOOK NOT TAKEN)
(ROM)
JSB = (RPGM)
BINARY PROGRAM ROUTINE
JSB =
ROMJSB
SAVE R0-R1 AND CALLING ROM#
JSB = (ROM)
ROM ROUTINE
RETURN TO CODE
SAD
RETURN TO KEYSRV
RETURN TO KYIDEL
R6
RETURN TO CODE
SAD
RETURN TO KEYSRV
RETURN TO KYIDLE
R0
R1
ROM#
RETURN TO ROMJSB
R6

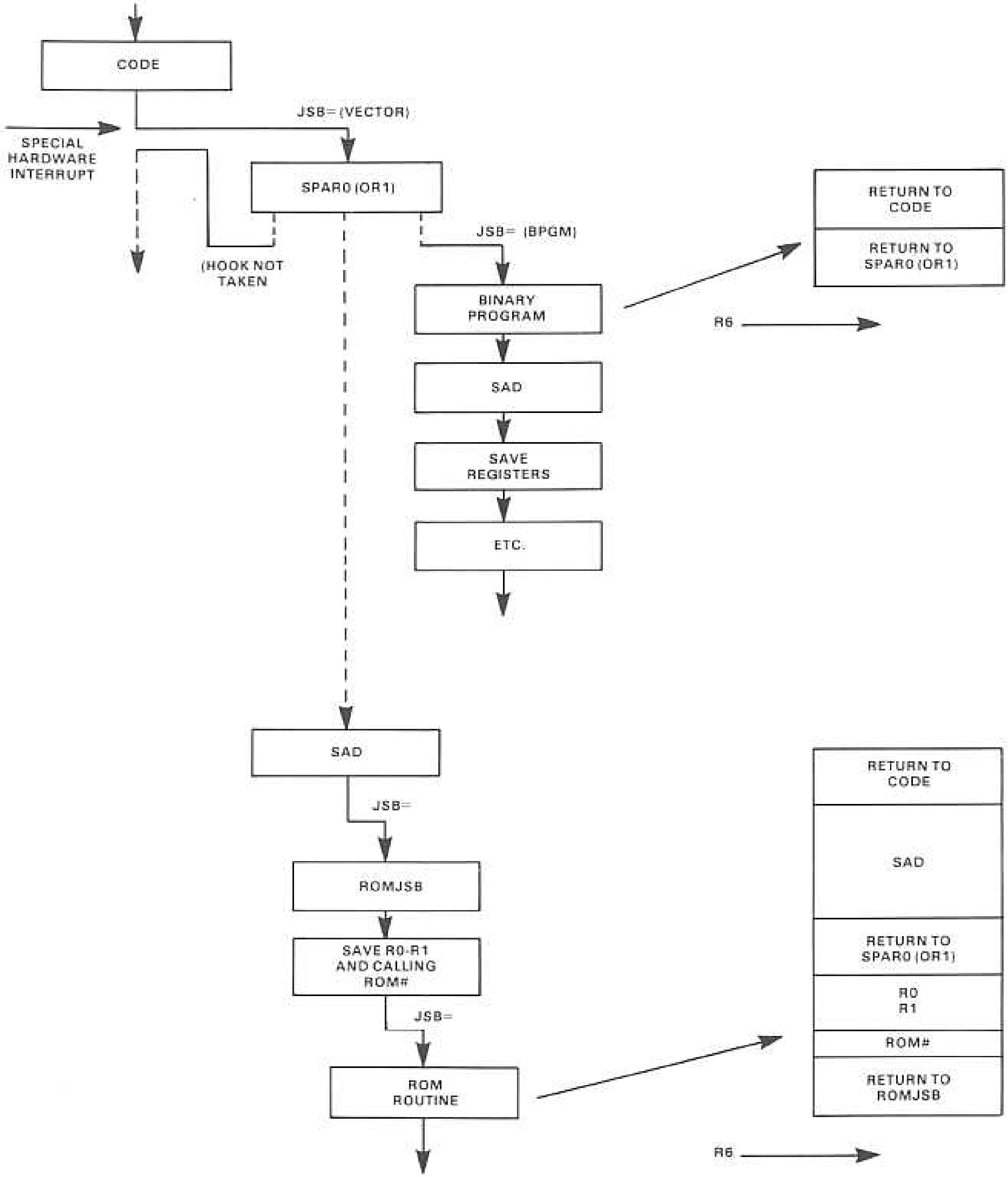
PRSIDL


EXEC
R17 =0 ?
N
Y
JSB=
XCBITS
JSB=
PARSER
JSB=
(HOOK NOT TAKEN)
PRSIDL
JSB= (BPGM)
BINARY PROGRAM ROUTINE
JSB= (ROM)
ROMJSB
SAVE R0-R1 & ROM#
JSB=
ROM ROUTINE
RETURN TO EXEC LOOP
RETURN TO XCBITS
RETURN TO PARSER
RETURN TO PRSIDL
R6
RETURN TO EXEC
RETURN TO XCBITS
RETURN TO PARSER
RETURN TO PRSIDL
R0
R1
ROM#
RETURN TO ROMJSB
R6

RMIDLE


EXEC
JSB=
RMIDLE
(HOOK NOT TAKEN)
(ROM)
JSB=
R17
=0
?
N
Y
R16
=0
?
N
Y
BINARY
PROGRAM
ROUTINE
JSB=
ROMJSB
SAVES R0-R1
AND ROM# OF
CALLING ROM
JSB=
ROM
ROUTINE
RETURN TO
EXEC LOOP
RETURN TO
RMIDLE
R6
RETURN TO
EXEC
RETURN TO
RMIDLE
R0
R1
ROM#
RETURN TO
ROMJSB
R6

SPARØ and SPAR1



SYSTEM RUNTIME TABLE

	ROUTINE	NAME	TOKEN	ATTRIBUTES
BTAB.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 FTSTA	STRING ARAY	40	0,3
	DEF JMPLB	THEN LABEL	41	0,207
	DEF P#ARAY	Array PRINT#	42	0,36
	DEF EJMPLB	ELSE LABEL	43	0,225
	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

ROUTINE	NAME	TOKEN	ATTRIBUTES
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 CHAIN.	CHAIN	116	0,241
DEF SECUR.	SECURE	117	0,241
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

ROUTINE	NAME	TOKEN	ATTRIBUTES
DEF PLOT.	PLOT	151	0,241
DEF PRNTR.	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 STORE.	STORE	166	0,141
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 INTEG.	INTEGER	177	0,323
DEF SHORT.	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 COM.	COM	205	0,324
DEF SKIPEM	DATA	206	0,320
DEF DEFFN.	DEF FN	207	0,312
DEF DIM.	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 UNSEC.	UNSECURE	220	0,141
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 LOAD.	LOAD	230	0,141
DEF REAL.	REAL	231	0,321
DEF RENUM.	REN	232	0,141
DEF SKIP!	!	233	0,241
DEF DEFA-.	DEFAULT OFF	234	0,241
DEF PEN.	PEN	235	0,241

ROUTINE	NAME	TOKEN	ATTRIBUTES
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 SKPLBL	STMT LABEL	257	0,3
DEF SQR5	SQR	260	20,55
DEF MIN10	MIN	261	40,55
DEF GTOLBL	GOTO LABEL	262	0,210
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 GSUB.	GOSUB LABEL	270	0,210
DEF COT10	COT	271	20,55
DEF CSEC10	CSC	272	20,55
DEF FTADR3	1-D ST ARAY	273	0,1
DEF EXP5	EXP	274	20,55
DEF INT5	INT	275	20,55
DEF LOGT5	LGT (10)	276	20,55
DEF LN5	LOG (E)	277	20,55
DEF FTADR4	2-D ST ARAY	300	0,1
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

ROUTINE	NAME	TOKEN	ATTRIBUTES
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 RESTL.	RESTORE LBL	335	0,227
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 USINL.	USING LABEL	345	0,327
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 FTADR1	1-D NUM ARY	363	0,1
DEF FTADR2	2-D NUM ARY	364	0,1
DEF TEST.	TEST KEY	365	0,341
DEF ERRORX	DUMMY	366	0,44
DEF INDEN.	INDENTATION	367	0,2
DEF ROM:GO	EXTERNAL ROM	370	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

Runtime Table/Tokens and Attributes for Graphics ROM #1

	ROUTINE	NAME	TOKEN	ATTRIBUTES
RUNTAB	DEF INIT	DUMMY # 0	0	
	DEF PLOTR.	PLOTTER IS	1	241
	DEF PRNTR.	PRINTER IS	2	241
	DEF CRT.	CRT IS	3	241
	DEF LIMIT.	LIMIT	4	241
	DEF GCLR.	GCLEAR	5	241
	DEF LOCAT.	LOCATE	6	241
	DEF BPLOT.	BPLOT	7	241
	DEF SCALE.	SCALE	10	241
	DEF SHOW.	SHOW	11	241
	DEF MSCAL.	MSCALE	12	241
	DEF CLIP.	CLIP	13	241
	DEF UNCLI.	UNCLIP	14	241
	DEF SETGU.	SETGU	15	241
	DEF SETUU.	SETUU	16	241
	DEF PENUP.	PENUP	17	241
	DEF GREAD.	BREAD	20	241
	DEF PEN.	PEN	21	241
	DEF LINET.	LINETYPE	22	241
	DEF PLOT.	PLOT	23	241
	DEF IPLOT.	IPLOT	24	241
	DEF MOVE.	MOVE	25	241
	DEF IMOVE.	IMOVE	26	241
	DEF DRAW.	DRAW	27	241
	DEF IDRAW.	IDRAW	30	241
	DEF RPLOT.	RPLOT	31	241
	DEF PDIR.	PDIR	32	241
	DEF BLOFF.	NOBLINK	33	241
	DEF AXES.	AXES	34	241
	DEF LAXES.	LAXES	35	241
	DEF GRID.	GRID	36	241
	DEF FRAME.	FRAME	37	241
	DEF LABEL.	LABEL	40	241
	DEF BLINK.	BLINK	41	241
	DEF LORG.	LORG	42	241
	DEF LDIR.	LDIR	43	241
	DEF CSIZE.	CSIZE	44	241
	DEF WHERE.	WHERE	45	241
	DEF CONTR.	CONTROL	46	241
	DEF CURSR.	CURSOR	47	241
	DEF DIGIT.	DIGITIZE	50	241
	DEF DUMMY	TRANSLATE	51	241
	DEF LGRID.	LGRID	52	241
	DEF GRAPH.	GRAPHICS	53	241
	DEF XAXIS.	XAXIS	54	241
	DEF YAXIS.	YAXIS	55	241
	DEF FXD.	FXD	56	241
	DEF ERRSC.	ERRSC	57	0,55
	DEF ERROM.	ERROM	60	0,55
	DEF RATIO.	RATIO	61	0,55
	DEF TAB.	TAB	62	20,45
	DEF LABEOL	LABEL EOLINE	63	35
	DEF PAGES.	PAGE SIZE	64	241
	DEF ALFAL.	ALPHA ALL	65	241
	DEF GRAFA.	GRAPH ALL	66	241
	DEF FRE.	FREE MEMORY	67	0,55

Runtime Table/Tokens and Attributes for Mass Storage ROM #320

	ROUTINE	NAME	TOKEN	ATTRIBUTES
RUNTIM	DEF INITIT	DUMMY # 0	0	241
	DEF ASSIG.	ASSIGN	1	241
	DEF MSCAT.	CAT	2	241
	DEF CHKOF.	CHECK READ OFF	3	241
	DEF CHECK.	CHECK READ	4	241
	DEF ERRORX	DUMMY ROUTINE	5	44
	DEF MSCPY.	COPY	6	241
	DEF MSCRE.	CREATE	7	241
	DEF INITI.	INITIALIZE	10	241
	DEF MSCHA.	CHAIN	11	241
	DEF MSLDB.	LOADBIN	12	241
	DEF MSLOD.	LOAD	13	141
	DEF MASSS.	MASS STORAGE IS	14	241
	DEF MSPRNT	PRINT#	15	241
	DEF ERRORX	DUMMY ROUTINE	16	44
	DEF ERRORX	DUMMY ROUTINE	17	44
	DEF MSPUR.	PURGE	20	241
	DEF READ.	READ#	21	241
	DEF MSREN.	RENAME	22	241
	DEF MSSTB.	STOREBIN	23	141
	DEF MSSTO.	STORE	24	141
	DEF PACK.	PACK	25	241
	DEF VOLUM.	VOLUME	26	241
	DEF GLOAD.	GLOAD	27	241
	DEF GSTOR.	GSTORE	30	241
	DEF ERROM.	ERROM	31	0,55
	DEF ERRSC.	ERRSC	32	0,55
	DEF TYP.	TYP	33	20,55
	DEF IS.	(VOLUME) IS	34	1,51
	DEF ERRORX	DUMMY ROUTINE	35	44
	DEF TO.	(RENAME) TO	36	1,51
	DEF RDNUM.	READ# NUMERIC	37	44
	DEF PRARR.	PRINT# NUM ARRAY	40	36
	DEF RDSTR.	READ# STRING	41	44
	DEF PRNUM.	PRINT# NUMERIC	42	36
	DEF PREOL.	PRINT# END OF LINE	43	35
	DEF PRSTR.	PRINT# STRING	44	36
	DEF RDARR.	READ# NUM ARRAY	45	44
	DEF PRARR$	PRINT# STRING ARRAY	46	36
	DEF RDARR$	READ# STRING ARRAY	47	44

8.7 Error Messages

Following is a list of the error messages provided by the Assembler ROM and the system monitor. For other errors refer to the owner's manual or to the manuals for other peripherals that may be attached to the HP-87.

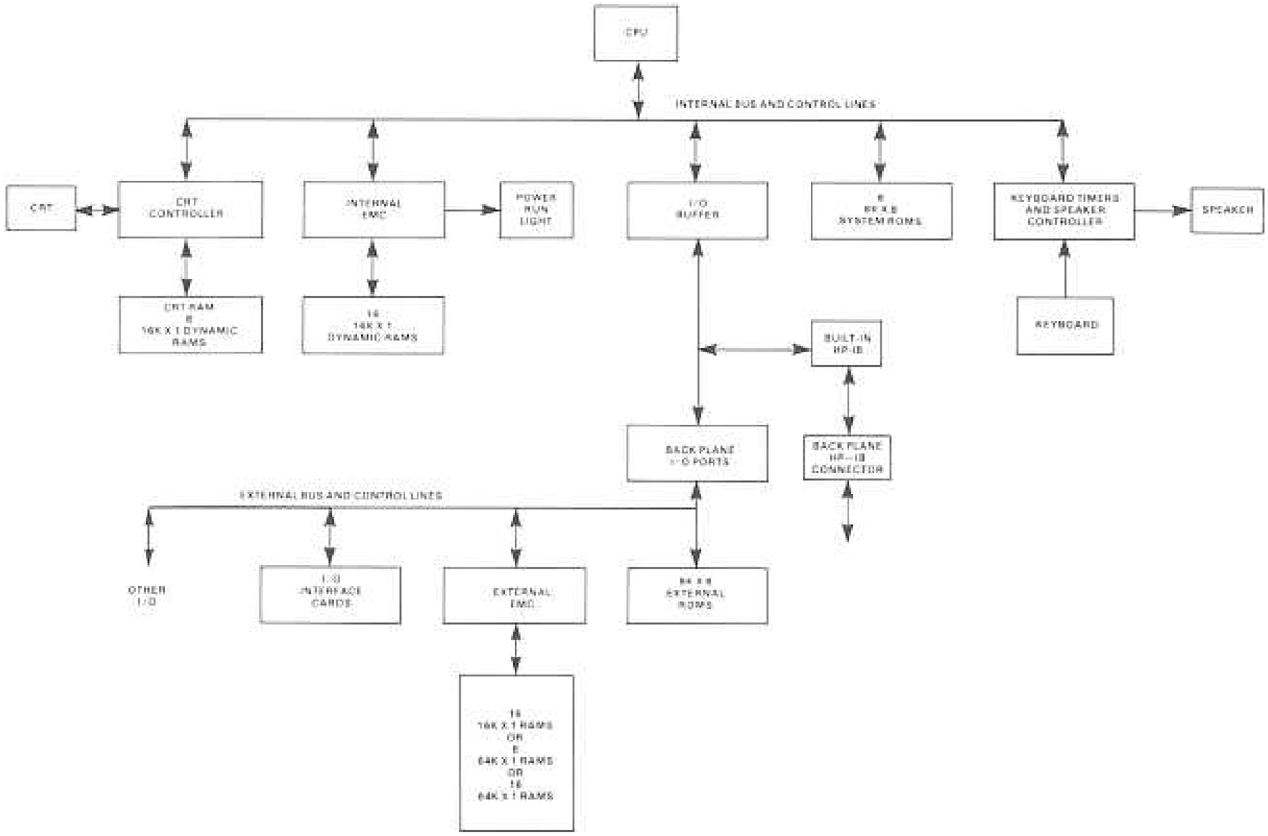
Assembler System Errors

ERROR 1Ø9:	ILL MODE	A command has been executed in the wrong operating mode (that is, ASSEMBLER has been typed when the computer is already in assembler mode).
ERROR 11Ø:	LBL	An invalid label has been seen; may have been longer than six characters or started with a digit.
ERROR 111:	OPCO	The opcode is not recognized; may have been misspelled, no space was typed between the 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; that is, 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 115:	ASSM ROM	At power-on, this means the ROM had a checksum error. At a breakpoint, any errors generated give this message.

Assembly Errors

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 the spcified line is out of range.
JMP TO	The jump to the specified line is out of range.
UND LAB	After assembly was completed, this label had not been defined in the program or in the global file.
ILL GLO	The GLO statement occurs after a NAM statement, ABS statement, or another GLO statement.

8.8 System Hardware Diagram



8.9 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 (that is, R32), R*, or R#.
AR	Address register. Can be register number, R*, or R#.
Literal	Literal value, up to 1Ø octal bytes in length. Can be BCD constant (that is, 99C), octal constant (that is, 12), or decimal constant (that is, 2ØD). Can also be specified by a label, where the literal quantity is a one- or two-byte value or address assigned to the label.
Label	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 sec.
B	Number of bytes.
T	Add one clock cycle if true (that is, the jump occurs).
R(x)	CPU register addressed by (x).
M(x)	Memory location addressed by (x) where (x) is a 16-bit address.
PC	Program counter stored in CPU registers R4 and R5. Used to address the instruction being executed.
SP	Subroutine stack pointer stored in 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.
$\leftarrow$	Is transferred to.
( )	Contents of.
$\overline{}$	Complement (that is, $\overline{x}$ is complement of x). This is one's complement if DCM=Ø and nine's complement if DCM=1.
$\cdot$	Logical "and."
v	Inclusive "or."
$\oplus$	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.

Instruction Format	Description	Addressing Mode	Opcode	Clock Cycles	Operation	Status											Binary/ BCD Option
											DCM = 0			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*, = Label	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 R0		001	3	ARP ← R0	—	—	—	—	—	—	—	—	—	—	—	
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 + AR + 1	X	X	X	X	—	—	X	X	—	X	0	Y
CMB *DR*, = *literal*	Compare byte	Lit. imm.	310	5	DR + M(PC + 1) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
CMBD *DR*, *AR*	Compare byte	Reg. dir.	330	6	DR + M(AR) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
CMBD *DR*, = *label*	Compare byte	Lit. dir.	320	6	DR + M(ADR) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
CMM *DR*, *AR*	Compare multi-byte	Reg. imm.	301	4 + B	DR + AR + 1	X	X	X	X	—	—	X	X	—	X	0	Y
CMM *DR*, = *literal*	Compare multi-byte	Lit. imm.	311	4 + B	DR + 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 + M(AR) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
CMMD *DR*, = *label*	Compare multi-byte	Lit. dir.	321	5 + B	DR + 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 R0		101	3	DRP ← R0	—	—	—	—	—	—	—	—	—	—	—	

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
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
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	—	—	—	—	—	—	—	—	—	—	—	
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	

Instruction Format	Description	Addressing Mode	Opcode	Clock Cycles	Operation	Status LSB	Status MSB	Status RDZ LDZ	Status Z	Status DCM	DCM = 0 E	DCM = 0 CY	DCM = 0 OVF	DCM = 1 E	DCM = 1 CY	DCM = 1 OVF	Binary/ BCD Option
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	
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	Reg. 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 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. indir.	341	5 + B	DR ← M(AR) AR ← AR + M	X	X	X	X	—	—	0	0	—	0	0	
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	

Instruction Format	Description	Addressing Mode	Opcode	Clock Cycles	Operation	Status											Binary/ BCD Option
											DCM = 0			DCM = 1			
						LSB	MSB	RDZ LDZ	Z	DCM	E	CY	OVF	E	CY	OVF	
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	—	—	—	—	—	—	—	—	—	—	—	
SBB *DR*, *AR*	Subtract byte	Reg. imm.	304	5	DR ← DR + AR + 1	X	X	X	X	—	—	X	X	—	X	0	Y
SBB DR, = *literal*	Subtract byte	Lit. imm.	314	5	DR ← DR + M(PC + 1) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
SBBD *DR*, *AR*	Subtract byte	Reg. dir.	334	6	DR ← DR + M(AR) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
SBBD *DR*, = *label*	Subtract byte	Lit. dir.	324	6	DR ← DR + 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 + AR + 1	X	X	X	X	—	—	X	X	—	X	0	Y
SBM *DR*, = *literal*	Subtract multi-byte	Lit. imm.	315	4 + B	DR ← DR + 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 + M(AR) + 1	X	X	X	X	—	—	X	X	—	X	0	Y
SBMD *DR*, = label	Subtract multi-byte	Lit. dir.	325	5 + B	DR ← DR + 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	

Instruction Format	Description	Addressing Mode	Opcode	Clock Cycles	Operation	Status: LSB	MSB	RDZ LDZ	Z	DCM	DCM = 0: E	CY	OVF	DCM = 1: E	CY	OVF	Binary/ BCD Option
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 complement byte	Reg. dir.	247	5 + B	DR → M(AR)	X	X	X	X	—	—	0	0	—	0	0	
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	
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 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. indir.	341	5 + B	DR ← M(AR), AR ← AR + M	X	X	X	X	—	—	0	0	—	0	0	
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	

8.1Ø Assembler Instruction Coding

7	6	5	4	3	2	1	0
0	DRP/ ARP	≠000001 =000001	Load with literal 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	000 001 010 011 100 101 110 111	BIN BCD SAD DCE ICE CLE RTN PAD	
1	0	1	000 001 010 011 100 101 110 111	REG REG LIT REG LIT INX LIT INX	IMM DIR IMM IND DIR DIR IND IND	Store/Load	M/B
1	1	0	00 REG 01 LIT 10 LIT 11 REG	IMM IMM DIR DIR	00 01 10 11	CMP ADD SUB 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

8.11 Keycode Table

DEC	KEYCODE OCT	KEY	DEC	KEYCODE OCT	KEY
0	0	ctrl@	48	60	0
1	1	ctrl A	49	61	1
2	2	ctrl B	50	62	2
3	3	ctrl C	51	63	3
4	4	ctrl D	52	64	4
5	5	ctrl E	53	65	5
6	6	ctrl F	54	66	6
7	7	ctrl G	55	67	7
8	10	ctrl H	56	70	8
9	11	ctrl I	57	71	9
10	12	ctrl J	58	72	:
11	13	ctrl K	59	73	;
12	14	ctrl L	60	74	<
13	15	ctrl M	61	75	=
14	16	ctrl N	62	76	>
15	17	ctrl O	63	77	?
16	20	ctrl P	64	100	@
17	21	ctrl Q	65	101	A
18	22	ctrl R	66	102	B
19	23	ctrl S	67	103	C
20	24	ctrl T	68	104	D
21	25	ctrl U	69	105	E
22	26	ctrl V	70	106	F
23	27	ctrl W	71	107	G
24	30	ctrl X	72	110	H
25	31	ctrl Y	73	111	I
26	32	ctrl Z	74	112	J
27	33	ctrl [	75	113	K
28	34	ctrl \	76	114	L
29	35	ctrl]	77	115	M
30	36	ctrl ^	78	116	N
31	37	ctrl _	79	117	O
32	40	SPACE	80	120	P
33	41	!	81	121	Q
34	42	"	82	122	R
35	41	#	83	123	S
36	44	$	84	124	T
37	45	%	85	125	U
38	46	&	86	126	V
39	47	'	87	127	W
40	50	(	88	130	X
41	51	)	89	131	Y
42	52	*	90	132	Z
43	53	+	91	133	[
44	54	>	92	134	\
45	55	—	93	135	]
46	56	.	94	136	^
47	57	/	95	137	_

8.12 Programming Hints

If execution of certain advanced programming ROM statements is attempted in assembler mode, unpredictable results can occur. These statements are:

- X REF L
- X REF V
- REPLACE VAR

INDEX