CHAPTER 1
PROGRAMS DEVELOPED IN ASSEMBLY LANGUAGE

TYPES OF PROGRAMS 1-1
  MAIN PROGRAMS 1-1
  SUBPROGRAMS 1-2
PROGRAM HEADERS 1-4
  PROGRAM TYPE BYTE 1-6
  DISPLACEMENT TO NAME-LENGTH BYTE 1-7
  DISPLACEMENT TO THE NEXT PROGRAM 1-7
  PROGRAM ENTRY ADDRESS 1-8
  NAME-LENGTH BYTE 1-8
  PROGRAM NAME 1-8
CARTRIDGE MEMORY 1-9

CHAPTER 2
SYSTEM SOFTWARE CONVENTIONS AND ROUTINES

FLOATING POINT SUBSYSTEM 2-1
  REPRESENTATION OF FLOATING POINT NUMBERS 2-1
  FLOATING POINT ROUTINE MEMORY USAGE 2-3
    Registers 2-3
    Floating Point Stack 2-4
  TYPES OF FLOATING POINT ROUTINES 2-6
    Arithmetic Routines 2-6
    Comparison Routine 2-6
    Polynomial Evaluation Routine 2-6
    Floating Point Stack Routines 2-6
    Exponential Function Evaluation Routines 2-7
    Trigonometric Function Evaluation Routines 2-7
    Random Number Routines 2-7
    Integer/Floating Point Conversion Routines 2-7
    Miscellaneous Floating-Point Operations 2-8
STRING-OPERATION SUBSYSTEM 2-8
DYNAMIC MEMORY MANAGEMENT SUBSYSTEM 2-9
DISPLAY CONTROL SUBSYSTEM 2-11
KEYBOARD ROUTINES 2-11
DATA ENTRY ROUTINES 2-12
GENERAL UTILITY SUBSYSTEM 2-13
  MEMORY PAGING ROUTINES 2-13
  BLOCK MEMORY MOVE ROUTINES 2-15
CARTRIDGE PROGRAM SEARCH ROUTINE 2-16
  RANDOM INTEGER ROUTINE 2-16
  SOUND GENERATION ROUTINE 2-16
  DEBUG MONITOR ENTRY ROUTINE 2-16
  POWER-UP, POWER-DOWN, AND BATTERY CHECK ROUTINES 2-17
I/O SUBSYSTEM 2-17
  PERIPHERAL ACCESS BLOCK (PAB) 2-18
  I/O CALLS 2-20
    OPEN Command (>00) 2-21
    CLOSE Command (>01) 2-26
    DELETE OPEN FILE Command (>02) 2-27
    READ DATA Command (>03) 2-28
    WRITE DATA Command (>04) 2-30

RESTORE command (>05) 2-31
DELETE Command (>06) 2-32
RETURN STATUS Command (>07) 2-33
FORMAT Medium Command (>0D) 2-35
READ CATALOG Command (>0E) 2-36
SET OPTIONS Command (>0F) 2-37
RESET Command (>FF) 2-38
USE OF THE IOS FROM BASIC 2-39

CHAPTER 3
RUNNING PROGRAMS IN THE BASIC ENVIRONMENT

BASIC OPERATING ENVIRONMENT 3-1
BASIC REGISTERS 3-2
BASIC RAM 3-4
System Reserved Area 3-5
Machine Language Subprogram Area 3-7
User Assigned String Table 3-8
Variable Name Table 3-8
Floating Point Stack 3-9
Free Memory Space 3-10
Dynamic Memory Area 3-11
Main Program Area 3-12
MEMORY USAGE AND SYSTEM REENTRY 3-12
LEVEL ZERO MEMORY USAGE 3-12
LEVEL ONE MEMORY USAGE 3-13
LEVEL TWO MEMORY USAGE 3-16
LEVEL THREE MEMORY USAGE 3-17
TRAP VECTOR MEMORY USAGE 3-13
SUBPROGRAM PARAMETER PASSING 3-19
SYNTAX CHECKING AND PARMETER PASSING 3-21
Checking for a Left Parenthesis 3-21
Advancing the Program Pointer 3-22
Calling the Appropriate Argument-Accessing Routine 3-22
Getting Further Arguments 3-23
Checking for the Right Parenthesis 3-23
Advancing the Program Pointer 3-23
Checking for the End-of-Statement Token 3-23
PASSING VARIABLES AS PARAMETERS 3-24
Simple Numeric Variables 3-25
GETNUM - Get a Numeric Value 3-26
GETADR - Get the Address of an Argument 3-27
ASSIGN--Changing the Value of a Numeric Variable 3-28
SIMPLE STRING VARIABLES 3-29
GETSTR - Get a String Argument 3-30
GETADR - Get the Address of a String Argument 3-31
ASSIGN--Changing the Value of a String Variable 3-32
String Creation and Deletion 3-32
ARRAY VARIABLES 3-34
Passing Elements of an Array 3-36
Accessing an Entire Array 3-37
Locating Elements in an Array 3-38
Value Assignment for an Array Element 3-39
PERIPHERAL I/O 3-40

OPENING DEVICES OR FILES 3-43
CLOSING DEVICES OR FILES 3-45
ERROR PROCESSING 3-46
ON WARNING PRINT 3-47
ON WARNING NEXT 3-47
ON WARNING ERROR 3-47
ON ERROR STOP 3-48
ON ERROR LINE NUMBER 3-48
REDIRECTION OF THE ERROR HANDLER 3-48
BREAKPOINT PROCESSING 3-49
MISCELLANEOUS SYSTEM ROUTINES 3-49

CHAPTER 4
ALPHABETIC LIST OF SYSTEM ROUTINES IN ROM

CHAPTER 5
ASSEMBLY LANGUAGE PROGRAMMING EXAMPLES

DIR: A WAFERTAPE DIRECTORY SUBPROGRAM 5-1
COPY: A FILE COPY SUBPROGRAM 5-9
NUMHEX:NUMBER TO HEXIDECIMAL STRING SUBPROGRAM 5-16
PRMYN: PROMPT FOR Y(ES) OR N(O) 5-20
SUBROUTINES FOR PERIPHERAL ACCESS 5-24
ADDING A PAB INTO THE LINKED LIST 5-25
STORING A PAB INTO DYNAMIC MEMORY 5-27
FINDING A PAB IN THE LINKED LIST 5-28
LOADING A PAB INTO THE TEMPORARY AREA 5-29
DELETING A PAB 5-30

CHAPTER 6
CC-40 SYSTEM HARDWARE DESCRIPTION

THE 70C20 PROCESSOR 6-2
MEMORY ORGANIZATION 6-4
GENERAL-PURPOSE REGISTER FILE 6-5
PERIPHERAL-REGISTER FILE 6-6
MEMORY IN THE CONSOLE 6-6
System RAM 6-7
System ROM 6-8
Processor ROM 6-9
CARTRIDGE MEMORY 6-9
Cartridge Memory Speed Control 6-10
PROCESSOR INTERRUPTS AND TIMER CONTROL 6-12
USE OF INTERRUPTS 6-12
USE OF THE TIMER AND EVENT COUNTER 6-15
CONSOLE INPUT/OUTPUT DEVICES 6-17
CONSOLE KEYBOARD 6-17
LIQUID-CRYSTAL DISPLAY (LCD) AND CONTROLLER 6-18
BEEPER 6-18
HEX-BUS-tm INTELLIGENT PERIPHERAL INTERFACE 6-19
POWER-ON HOLD REGISTER OUTPUT 6-20
LOW BATTERY VOLTAGE INPUT 6-20

CHAPTER 7
CC-40 ASSEMBLY-LANGUAGE INSTRUCTION SET

INSTRUCTION LINE FORMAT 7-1
LABELS 7-1
OP CODES AND OPERANDS 7-2
COMMENTS 7-3
INSTRUCTION DATA SHEETS 7-3
INSTRUCTION DESCRIPTION 7-4
INSTRUCTION TABLE ENTRIES 7-5
PROCESS DESCRIPTION ENTRIES 7-6
FLAG-DESCRIPTION ENTRIES 7-6
CONVENTIONS AND SYMBOLS USED IN DATA SHEETS 7-6
NUMBER REPRESENTATION CONVENTIONS 7-6
GENERAL-PURPOSE REGISTER SYMBOLS 7-7
PERIPHERAL-FILE REGISTER SYMBOLS 7-8
SPECIAL SYMBOLS 7-8

CHAPTER 1
PROGRAMS DEVELOPED IN ASSEMBLY LANGUAGE

When the Editor/Assembler cartridge is used to develop machine-language programs, these programs must follow conventions appropriate to CC-40 hardware and software design. They must, for example, be developed as one of two specific types of programs. They must also be developed for inclusion in linked lists of other programs and subprograms. If they are designed for inclusion in cartridges, they must be accounted for in cartridge-header design.

TYPES OF PROGRAMS

The Editor/Assembler cartridge can be used to develop two types of CC-40 machine-language programs: main programs and subprograms. Main programs differ from subprograms in the following ways.

* The way they are loaded into memory
* Their location in memory
* The way they are run
* The way they stop running.

MAIN PROGRAMS

Only one main program can be present in system RAM. If a subsequent main program is loaded, it replaces the one previously loaded.

A main program is loaded into system RAM by an OLD command. The syntax of the command is OLD "device.filename". The device designation is the device number of the peripheral unit from which the file is loaded, and the file name is the name of the

file as it is stored on the peripheral. The file must consist of a single record that contains the entire program in relocatable format.

Loading a program with the OLD command places it at the high end of system RAM, overwriting any previous main program located there. The program remains in the main program space until a NEW or NEW ALL command is executed, until another program is loaded, or until some other circumstance causes complete system initialization.

A main program in system RAM is executed by a RUN command with no argument. A RUN command with an argument can be used to both load and execute a main program with a single command. The syntax of the statement is RUN "device.filename".

A main program terminates execution by returning control to the BASIC operating system. Control is returned to BASIC through entry-point routines in system ROM. One of three entry points, RETSYS, RETNEW, or RETINT, is used. The current state of system RAM determines which entry point should be used. (See chapter * for details.)

SUBPROGRAMS

While only one main program can be present in system RAM, multiple subprograms can be present. Subprograms are loaded by the LOAD subprogram in system ROM. The syntax of the statement which accesses the subprogram is CALL LOAD("device.filename"); the file named may contain one or more subprograms. The file must consist of a single record that contains the subprogram(s) in relocatable format. When the file contains more than one

subprogram. the subprogram headers (see below) must form a linked list. The last subprogram in the file (loaded highest in memory) must be first in the linked list. The last subprogram in the list must have a "zero" link to terminate the list.

The LOAD subprogram places a subprogram in the machine-language subprogram area of memory. The subprogram area is located in the low end of system RAM between the system reserved area and the user assigned string area. If the area contains previously loaded subprograms, subsequently loaded subprograms are linked to the end of the list. Subprograms remain in this area until a NEW ALL command is executed or until some other circumstance causes complete system initialization.

Loaded subprograms are executed by the CALL command from the command level of BASIC or by a CALL statement in a BASIC proram. The general form of the CALL statement is CALL subprogram-name[argument-list], where the subprogram name is the name included in the subprogram header, and the argument-list contains information to be used in the subprogram.

The argument list in the CALL command or statement is optional. It is not required by the CALL, but it may be required by the subprogram. When an argument list is required by the subprogram, the argument list has a syntax defined in the subprogram. The process of passing information from the CALL statement to the subprogram is described in the chapter 3.

The name of a subprogram must be chosen with care. When the system executes a CALL command or statement, it must search for the subprogram through four linked lists. It executes the first subprogram with the specified name. Any subsequent subprogram

with the same name is never found. The linked lists, in the order in which they are searched, are as follows.

1) Subprograms that are built into the system ROM, such as LOAD, PEEK, and POKE

2) Subprograms developed through assembly language which are loaded by LOAD into the subprogram area of system RAM

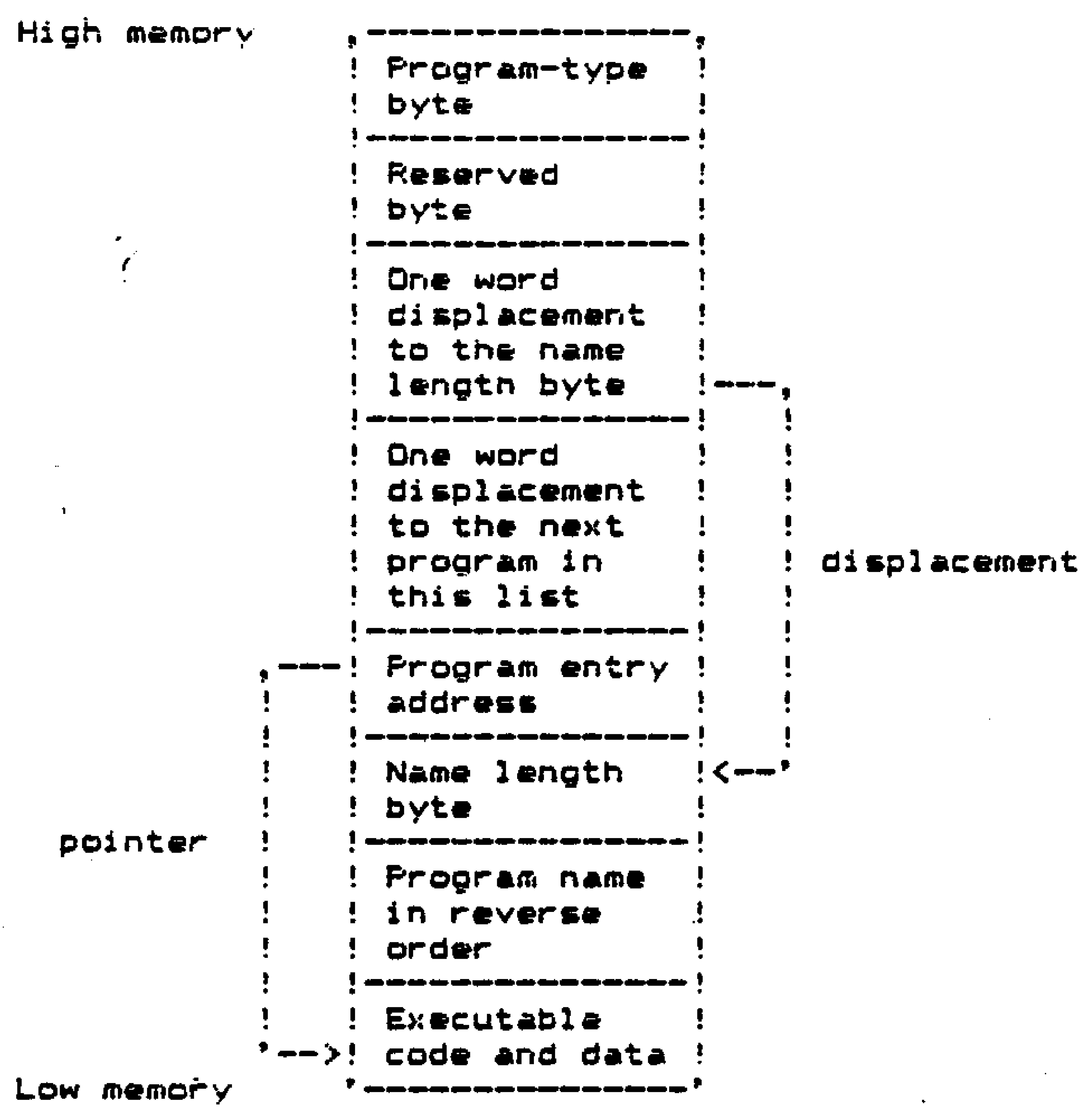
3) BASIC subprograms in system RAM; that is those which are part of a BASIC program that is running in the main program area

4) Any subprograms or programs, in BASIC or machine language, in cartridge memory

A subprogram returns control to the calling program by executing the machine language code for a RETS instruction. However, subprograms which are designed to be called only from the command level of the system may use the same system entry points that are used by main programs.

PROGRAM HEADERS

Every main program and subprogram developed in assembly language must have a program header. A header provides information necessary for the system to load, find, and execute a program or subprogram. The header is located at the high end of the program. It has the following form.



The following example shows the assembly-language listing of the code for a header for subprogram.

```
0012 0017         NAMLOW EQU $
0013 0017    50
     0018    45
     0019    45
     001A    42   RTEXT 'BEEP'          Name in reverse order
0014        001B' NAMHGH EQU $
0015 001B    04   BYTE   NAMHGH-NAMLOW  Name length byte
0016 001C 0000'   DATA   BEEP           Program entry point
0017 001E 0000    DATA   0              Displacement to next
0018 0020 FFFC    DATA   NAMHGH-$+1     Displacement to name
0019 0022    00   BYTE   0              Reserved byte
0020 0023    44   BYTE   >44            Program type byte
```

PROGRAM TYPE BYTE

The program-type byte, located at the highest address in the header, identifies four important characteristics of the program or subprogram, as shown in the following table.

BIT	VALUE	SIGNIFICANCE
7	1	main program
	0	subprogram
6	1	machine language
	0	BASIC language
5	0	reserved
4	0	reserved
3	1	located in cartridge memory
	0	located in system RAM
2	1	relocatable format
	0	absolute format
1	0	reserved
0	0	reserved

The type byte in the header example is set for a relocatable machine-language subprogram. The location information in bit 3

is used only when the program is in BASIC language. The format information in bit 2 is used only when the program is in machine language.

DISPLACEMENT TO NAME-LENGTH BYTE

The displacement to the name-length byte is used by the system when it searches the linked list for a program or subprogram. This displacement is the additive offset from the address preceding the displacement word to the address of the name-length byte. As the example shows, the displacement word is calculated by the following expression.

address of name-length byte - current address + 1

A dollar sign ($) is used in an assembly to specify the vlue of the current address.

DISPLACEMENT TO THE NEXT PROGRAM

The displacement to the next program links programs and subprograms into lists. The system searches through these linked lists for a particular program or subprogram. The displacement is the additive offset from the address preceding the displacement word to the address of the program-type byte in the next header in the list. The displacement is calculated at the time of assembly by the following expression.

address of next header - current address + 1

In the expression above, the address of next header is the address of the program-type byte for the next program header. A displacement value of zero, as shown in the header example, is used to terminate a linked list of headers.

PROGRAM ENTRY ADDRESS

The program entry address is the address at which program execution is to begin. This address is usually the lowest byte of executable code, but for special applications it may be set to any point within the program.

NAME-LENGTH BYTE

The name-length byte is the number of characters in the program name. As shown in the example, the name length can be computed during assembly through the proper use of labels before and after the name. Although the example shows the name-length byte and name as a part of the program header, these two items are not required to be in the header. The name length and name may be located anywhere in memory as long as the length byte appears at the highest address and the name precedes it in reverse order.

PROGRAM NAME

The program name is the string of characters which the system examines when it searches for a program. The name must be stored in reverse order immediately preceding the name-length byte. As shown in the example, text can be stored in reverse order automatically by the RTEXT assembler directive. A program name may contain up to 15 characters. The first character must

be an uppercase alphabetic character or an underline. The remaining characters may be uppercase alphabetic characters, numeric digits, or underlines.

CARTRIDGE MEMORY

In addition to its use as an extension of system RAM, cartridge RAM can be used to store and run machine language programs and subprograms. There are minor differences between loading and running programs in cartridge memory and loading and running them in system RAM.

To load programs and subprograms into cartridge RAM and to create and maintain the necessary linked list of headers, utility subprograms named CARTINIT and CARTLOAD must be loaded into system RAM (see "Cartridge Loader" in chapter 7 of the CC-40 Editor/Assembler User's Guide).

Unlike system RAM, cartridge RAM permits more than one main program to be present. In cartridge memory, all available programs and subprograms are linked together in a single linked list of program headers. Although no specific ordering is required, the normal convention is that the main programs appear in the list before the subprograms.

Main programs in a cartridge are executed by a RUN statement which includes the program name. The syntax of the statement is RUN "program-name", where the program name is pointed to through the displacement in the program header. Similarly, subprograms in a cartridge are executed by a CALL statement which includes the subprogram name.

Each cartridge must contain a cartridge header in the twelve bytes which begin at address >9000 of page zero. The header contains the following information.

Address	Information
>9000->9001	Expected value >A55A
>9002	Size information / Version number
	Bit 7 = 0 : 8K cartridge
	= 1 : 16K cartridge
	Bit 6 = 0 reserved
	Bits 5-0 = Cartridge version number
>9003	Execution and speed information
	Bit 7 = 0 : no immediate execution program
	= 1 : execute first program in list immediately on powerup
	Bits 6-0 = Speed control in tenths of microseconds
>9004->9005	Pointer to first program header
>9006->9007	Reserved word: always >0000
>9008->9009	Pointer to next available address between >9000 and >CFFF
>900A->900B	Pointer to next available address between >5000 and >8FFF

The expected value in the cartridge header permits the system to verify that a cartridge is present.

Bit 7 at address >9002 indicates whether the cartridge contains 8K of RAM or 16K of RAM. Bit 6 of that location is reserved and is reset to zero. The remaining 6 bits can be used for a version number

When bit 7, at address >9003, is set to ONE, it indicates that the cartridge contains a main program designed for immediate execution after CC-40 power up or reset. This program is the first program in the linked list. The immediate-execution program must be a main program. Specification of a subprogram will generate a "Bad program type" error.

Bits 0 through 6 at address >9003 control memory-access time. For cartridge RAM, which can be accessed at processor clock rate, this value should be ?000011. Cartridge RAM may be run at slower speeds, if, for example, the RAM is used to simulate slow-access ROM.

The value in these six bits represents memory access time in tenths of microseconds, but within the BASIC operating environment only certain ranges of values are used. Values between ?000100 and ?010010 cause the cartridge to be accessed at the same rate that system ROM is accessed with (the processor clock frequency divided by 7). Larger values in these bits cause both cartridge memory and system ROM to be accessed at the processor clock rate divided by 9. See Chapter 6 for details about memory access times.

The two-byte word at address >9004 is a pointer to the program-type byte in the header of the first program or subprogram in the cartridge. The CARTINIT subprogram is used to load a cartridge header into the cartridge memory. If no program is loaded with the cartridge header, this word should be initialized to zero. If one or more programs are loaded along with the cartridge header, this pointer must be initialized to the first header. After the initial load into the cartridge, the CARTLOAD subprogram can be used to load additional programs and subprograms into the cartridge. The linked list is automatically updated by CARTLOAD.

The word at address >9006 is reserved. It must contain a value of zero.

The word at address >9008 points to the next available loading location above the header. If the cartridge header is loaded alone, this word should point to the address following the header. If one or more programs are loaded with the cartridge header, this word should point to the address following the last program. The CARTLOAD utility automatically updates this pointer.

The word at address >900A points to the next available loading location below the header. When the cartridge header is loaded into an 8K RAM cartridge, this word should be initialized to zero. When the cartridge header is loaded into a 16K RAM cartridge, this word should be initialized to >5000. The CARTLOAD utility also updates this pointer.

CHAPTER 2
SYSTEM SOFTWARE CONVENTIONS AND ROUTINES

The software in CC-40 system ROM contains a BASIC interpreter and operating routines which support input to and output from BASIC. This software is organized into a number of subsystems, and each subsystem contains a number of machine-language subroutines which may be used within programs and subprograms developed in assembly language.

FLOATING POINT SUBSYSTEM

The floating point subsystem consists of a group of system subroutines that can be used to manipulate floating point numbers. These routines perform tasks which range from simple arithmetic operations, such as addition and subtraction, to function evaluations, such as sine, exponential, and logarithm evaluation. The subsystem also includes a variety of routines for manipulation of floating point numbers in the register file and system RAM.

REPRESENTATION OF FLOATING POINT NUMBERS

The floating point subsystem uses base 100 representation for floating point numbers. Each floating point number requires eight bytes of memory. The first byte (at the lowest address) is used to represent a base 100 exponent and an algebraic sign of the number. The remaining seven bytes represent the mantissa of the number.

The base 100 exponent is represented as a 7-bit hexadecimal value biased by >40 (64). The resulting range of the exponent is from -64 to +63. The following table shows the correspondence

between the biased hexadecimal value and the exponent of the number.

Biased hexadecimal exponent	>00	to	>40	to	>7F
Base 100 exponent	-64	to	0	to	+63
Base 10 (decimal) exponent	-128	to	0	to	+126

The most significant bit of the exponent byte determines the algebraic sign of the floating point number. If the number is negative, the entire exponent byte is inverted (one's complement), thus setting the most significant bit to one. The result is that the most significant bit of the exponent byte is set to one for all negative numbers and reset to zero for all positive numbers.

Each byte of the mantissa contains a base 100 digit between 0 and 99 represented in binary coded decimal (BCD) format. In other words, each four-bit nibble contains a decimal digit between 0 and 9. The resulting decimal precision is 13 or 14 digits, depending upon the contents of the first byte of the mantissa. The first byte of the mantissa (at the lowest address) contains the most significant base 100 digit. The mantissa is always normalized: the implied position of the decimal point is immediately following this most significant digit.

The exponent and mantissa combine to provide a decimal numeric range from -9.9999999999999E+127 through -1.E-128; zero; and +1.E-128 through +9.9999999999999E+127.

The number zero does not follow the rules just defined for floating point numeric representation. When the first byte of the mantissa is zero, the entire floating point number is

considered to be zero, regardless of the values in the other seven bytes.

The following examples show some decimal numbers and their corresponding base 100 and floating point representations.

Decimal value	Base 100	Representation (low to high)
0	$0*100^{xx}$	>xx >00 >xx >xx >xx >xx >xx >xx
1	$1*100^{0}$	>40 >01 >00 >00 >00 >00 >00 >00
-1	$-1*100^{0}$	>BF >01 >00 >00 >00 >00 >00 >00
10	$10*100^{0}$	>40 >10 >00 >00 >00 >00 >00 >00
100	$1*100^{1}$	>41 >01 >00 >00 >00 >00 >00 >00
1000	$10*100^{1}$	>41 >10 >00 >00 >00 >00 >00 >00
10000	$1*100^{2}$	>42 >01 >00 >00 >00 >00 >00 >00
1.245	$1.245*100^{0}$	>41 >01 >24 >50 >00 >00 >00 >00
123456789188	$12.3456789188*100^{5}$	>45 >12 >34 >56 >78 >91 >88 >00
.00102	$10.2*100^{-2}$	>3E >10 >20 >00 >00 >00 >00 >00
.01	$1*100^{-1}$	>3F >01 >00 >00 >00 >00 >00 >00

FLOATING POINT ROUTINE MEMORY USAGE

Floating point routines use both registers and system RAM for obtaining and storing floating point numbers.

Registers

Floating point routines use an area of the register file called the function level temporary area to perform most operations. The area is from register >58 through register >7F. Three key buffers within this area are the floating point accumulator (registers >75 through >7D), the floating point

C ?

second argument (registers >6B through >72), and the floating point status byte (register >7F). The function level area is all the memory that is required for most of the lower level floating point routines, such as addition and division.

Floating Point Stack

Higher level floating point routines require the use of a floating point stack in system RAM. The floating point stack is defined by three pointers.

A pointer to the base address of the floating point stack is maintained in the system reserved area of RAM at addresses (>8EA,>8EB). System initialization sets this pointer to the highest addressed byte of the variable name table area. The next higher addressed byte is the first byte of the floating point stack.

A pointer to the address of the last byte of the last value which has been pushed onto the floating point stack is maintained in register pair (>56,>57). The stack pointer is initialized to the same value as the base address pointer. Thereafter, the stack pointer is always assumed to be pointing to the highest addressed byte of the stack. The stack, therefore, is considered empty whenever the value of the stack pointer is equal to the value of the base address pointer.

A pointer to free system memory space is maintained in register pair (>54,>55). This free space pointer determines the highest address which can be used by the floating point stack. (Another function of this pointer is to delimit the lowest end of the dynamic memory management area, as described below.)

A stack entry consists of the eight bytes that make up a floating point number. The stack is pre-incrementing. Pushing a value onto the stack includes (1) incrementing the stack pointer by eight, (2) verifying that the stack pointer has not exceeded the free space pointer, and (3) copying the eight byte stack entry (from low to high) into the eight bytes just added to the stack. Similarly, popping a value from the stack includes (1) verifying that the stack pointer does not equal the base address, (2) copying the eight byte entry from the top of the stack--the byte indicated by the pointer and the next seven lower addressed bytes, and (3) decrementing the stack pointer by eight.

Routines are provided in the floating point subsystem for pushing a value from the floating point accumulator onto the stack and for popping a value from the stack into the floating point accumulator or into the floating point second argument.

The only other requirement of the floating point subsystem involves the handling of errors. The higher level floating point routines report argument errors by loading the A register with an error code and executing a specific TRAP instruction. This instruction transfers control to a branch vector in the system reserved area at addresses >81C through >81E. If the BASIC operating environment has not been altered, this branch vector transfers control to the BASIC error handler. However, the BASIC error handler is not appropriate for machine-language programs operating outside the BASIC environment. Such programs must initialize the branch vector to their own error handlers. (See "Error Processing" in chapter 3.)

TYPES OF FLOATING POINT ROUTINES

Floating point routines range from those which perform arithetic operations to those which perform utility and housekeeping operations.

Arithmetic Routines

Floating point arithmetic routines, described completely in the alphabetized list of descriptions in chapter 4, include the following.

Floating point addition (FADD)
Floating point subtraction (FSUB)
Floating point multiplication (FMUL)
Floating point division (FDIV)

Comparison Routine

System ROM includes one floating point comparison called FLTCMP.

Polynomial Evaluation Routine

One polynomial evaluation routine called POLYX is included in system ROM.

Floating Point Stack Routines

Three floating point stack routines in system ROM include the following.

Floating point stack push from floating point accumulator (FPPUSH)
Floating point stack pop into floating point accumulator (FPPOP)
Floating point stack pop into second argument (FCPARG)

Exponential Function Evaluation Routines

The four routines in system ROM which calculate the value of exponential functions are as follows.

Exponential function (EXP)
Natural logarithm function (LN)
Common logarithm function (LOG)
Square root function (SQR)

Trigonometric Function Evaluation Routines

Four routines in system ROM which calculate the value of trigonometric functions are as follows.

Arccosine function (ACOS)
Arcsine function (ASIN)
Arctangent function (ATN)
Cosine function (COS)
Sine function (SIN)
Tangent function (TAN)

Random Number Routines

Two routines in System ROM are used in generating a series of random floating point numbers.

Generate a random number in the Range 0 <= N < 1 (RND)
Reset the random number generator (RSTRND)

Integer/Floating Point Conversion Routines

System ROM contains two routines to convert between integer and floating point representations of values.

Convert floating point value to integer value (CFI)
Convert floating point value to unsigned integer value (CFIUNS)
Convert integer value to floating point value (CIF)

CFIUNS ?

Miscellaneous Floating-Point Operations

Eleven additional routines in ROM, listed below, perform utility and housekeeping operations.

- Clear the floating point second argument (CLRARG)
- Clear the floating point accumulator (CLRFAC)
- Clear the non-argument portion of the function level area (CLRRES)
- Copy the floating point accumulator to the second argument (COPYFA)
- Divide by zero warning (DIVZER)
- Load the floating point second argument (GCONA)
- Floating point greatest integer function (GRINT)
- Floating point normalization (NORMAL)
- Overflow warning (OVFLOW)
- Floating point rounding (ROUND, ROUND2)
- Swap the floating point accumulator and the second argument (SWAPFA)

STRING-OPERATION SUBSYSTEM

A string is any sequence of ASCII character codes. In the CC-40 system, strings are stored as a one byte string length preceded by the sequence of character codes in reverse order. A pointer to a string points to the length byte. For example, the string "HELLO" takes the form:

```
            ,-----------------------------------,
low address ! >4F ! >4C ! >4C ! >45 ! >48 ! >05 ! high address
            '-----------------------------------'
```

System ROM provides five routines for string comparison and for conversion to and from other forms of data representation. These routines are as follows.

- Comparison routine (STRCMP)
- Convert integer to string (CIS)
- Convert string to integer (CSI)
- Convert floating point number to string (CNS)
- Convert string to floating point number (CSN)

DYNAMIC MEMORY MANAGEMENT SUBSYSTEM

The dynamic memory management subsystem is composed of four routines which can be used to allocate and deallocate blocks from an area of memory that is delimited by four system pointers.

The base address of the dynamic memory area is maintained by a pointer stored in the system reserved area at addresses >BEC,>BED. This value points to the first byte below the current program image in the main program area.

A free space pointer is maintained in register pair >54,>55. This value points to the first free byte below the lowest allocated block in the dynamic memory area. A major function of this pointer is to indicate the highest addressed byte which may be used by the floating point stack.

The floating point stack pointer is maintained in register pair (>56,>57). The stack pointer always points to the highest addressed byte of the stack. The stack pointer also indicates the low end of the lowest block of available dynamic memory: the block bounded by the free space pointer and the stack pointer.

The fourth pointer is the first-free-block pointer located at addresses >BE6,>BE7 in the system reserved area. This value points to the first (highest addressed) free block of memory in the dynamic memory area. Other free blocks are maintained in a linked list.

Initialization of dynamic memory management sets the free space pointer and the first-free-block pointer to the same value as the base address pointer. Dynamic memory allocation uses a first-fit algorithm: The first available block of sufficient size is reduced by the requested number of bytes (plus two bytes for overhead information). The remainder of the block (if any)

is reentered in the linked list of free blocks. The search for a free block is linear, always beginning with the highest addressed block (indicated by the first free block pointer), and ending at the lowest addressed block (the one delimited by the free space pointer and the stack pointer). When a block is deallocated, it is linked into the list at the appropriate position depending upon its address. If it is immediately adjacent to any other free block(s), the deallocated block and the adjacent free block(s) are listed as a single free block.

The four dynamic memory management routines consist of two allocation routines (MNEW, STGNEW) and two deallocation routines (MDISP, STGDSP). The STGNEW and STGDSP routines are used to manipulate blocks of 255 bytes or less. These two routines must be used in conjunction with one another and are used by BASIC almost exclusively to handle string values. The MNEW and MDISP routines also must be used together and can process blocks of any size from 1 to 32767 bytes. These two routines are used by BASIC to handle all structures other than string values.

An additional routine, (FRE) can be used to determine the total amount of free space, the largest available block of free space, and the number of individual blocks of free space.

The five routines which manage the dynamic memory area are as follows.

Memory allocation routines (MNEW, STGNEW)
Memory deallocation routines (MDISP, STGDSP)
Free memory information routine (FRE)

DISPLAY CONTROL SUBSYSTEM

The display control subsystem consists of a set of primitive (low-level) routines that manage the liquid crystal display. These routines turn the cursor on and off, position the cursor, clear the display, and write ASCII characters to the display. Also, a routine is provide to update the display indicators from indicator status bytes that are maintained in the system reserved area.

The system uses register >53 to maintain the current cursor position relative to the start of the display. Cursor positions range from 0 to 30. The display control routines ignore a cursor position greater than 30 and place the cursor in the last position. Only the display character, display buffer, and clear display routines use memory other than register >53 and the A,B register pair. These routines use only a few registers in the function level temporary area.

The nine routines listed below make up the system used to manage the display.

- Clear the entire display (CLRDSP)
- Display contents of buffer (DSPBUF)
- Display a single character (DSPCHR)
- Initialize the display hardware (DSPINT)
- Turn off the cursor (OFFCRS)
- Turn on the flashing block cursor (ONCRS)
- Set the cursor position (SETCRS)
- Turn on the non-flashing underline cursor (UNCRS)
- Update the display indicator status (WRTIND)

KEYBOARD ROUTINES

The CC-40 Keyboard is a software scanned device. The system provides three routines for the purpose of reading the current status of the keyboard. One routine reads the status of the

keyboard and returns the appropriate information. A second routine continually reads the keyboard until a key is pressed and then returns the appropriate information. The last routine checks whether the BREAK key is being pressed.

In addition to the A,B register pair, the keyboard routines use the low end of the function level temporary area. They also use a specific part of the system reserved area (>834 to >839) to save important information between successive scans of the keyboard. Finally, they modify locations >83A and >83B as needed to maintain the status of the keyboard indicators--SHIFT, CTL, FN, and UCL.

The three routines which make up the keyboard subsystem are listed below.

Read the keyboard status (KSTAT)
Wait for a single-key entry (KEYIN)
Check the BREAK key (CHKBRK)

DATA ENTRY ROUTINES

Data entry routines are provided for the purposes of accepting input from through the keyboard and maintaining the display. These routines use an 80-byte buffer in the system reserved area from >83E to >88D. They use the keyboard and display control routines to display the current contents of the buffer, accept user input from the keyboard, modify the contents of the buffer, and update the display from the buffer. Additional routines are provide to clear the input buffer and to copy the contents of the input buffer into the playback buffer (locations >928 to >977 of the system reserved area).

The four routines used for keyboard input, display, and buffer management are as follows.

?

Accept and display input from keyboard (DUFIN, LINEIN)
Clear the input buffer (CLRINF)
Copy contents of input buffer to playback buffer (M[illegible]INPB)

?

GENERAL UTILITY SUBSYSTEM

A group of routines in system ROM is available for a variety of uses such as accessing memory, moving blocks of data, sounding the beeper, and accessing the Debug Monitor.

MEMORY PAGING ROUTINES

The system provides four paging routines for the purpose of transferring program control from one page of the system or cartridge memory to any other page of the system or cartridge memory. These routines require branch tables to be present at specific locations in each page of memory. Branch tables in cartridge memory begin at address >9000. (Because page zero of cartridge memory, however, must contain a cartridge header in address >9000 through >900B, the page zero branch table can contain useful entries only at and above >900C.)

A branch table is simply a list of branch vectors to each of the routine entry points on a page that are to be accessed other pages of memory.

Paging routines include two pairs of routines--one pair used for system memory (BRPAG, CALPAG) and the other pair for cartridge memory (STERP, CTERP). Within each pair, one routine (BRPAG or STERP) is used to branch to another page of memory with

no provision for returning to the current page. The other routine (CALPAG or CTERP) is used to call subroutines that will return control to the calling program.

All four routines require the same input--the memory page number (0 to 3) in the A register and the branch table offset in the B register. The branching routines select the requested page number, add the base address of the table (>9000 for cartridge memory) to the branch table offset, and transfer control to the resulting address in the selected page. The branch vector at that address transfers control to the entry point of the routine in memory.

The calling routines do more than the branching routines. Since execution is expected to return, the calling routines must save the memory page number and return address. The assembly-language CALL instruction that results in a transfer of control to the paging routine automatically places the return address on the processor stack in the register file. The paging routine reads the current memory page number and pushes it on the processor stack as well. Then, the routine selects the destination page, adds the table base address to the table offset, and calls the resulting address. This last call places the return address of the _paging routine_ on the processor stack. Thus, a paged call requires five bytes of stack instead of the two bytes required to call a subroutine within the same page of memory. When the subroutine executes the machine code from a RETS instruction, control returns to the paging routine. The paging routine pops the return page number from the stack.

restores the page to the previous one and execution returns to the calling program at the last address on the register-file stack.

The four routines for branching and calling among pages of system and cartridge memory are the following.

Branch to system memory page (BRPAG)
Call to system memory page (CALPAG)
Branch to cartridge memory page (BTERP)
Call to cartridge memory page (CTERP)

BLOCK MEMORY MOVE ROUTINES

Two memory move routines are used to copy the contents of a block of memory from one location to another. The registers used by these routines are in the function level temporary area and are the same for both routines. Each routine requires specification of a source pointer, a destination pointer, and a block size (number of bytes to be moved). The only differences in the routines are the direction of the move and the ends of the blocks to which the pointers are initialized.

Since a source block and a destination block may sometimes overlap, separate routines are provided for moving memory from low-to-high and from high-to-low addresses. The move-down routine is used to copy a block to a lower address. The block pointers must therefore initially point to the low ends of their source and destination blocks. The move-up routine is used to copy a block to a higher address, and the pointers point to the high ends of the two corresponding blocks.

The two memory-move instructions are as follows.

Move memory contents from high address to low address (MOVDWN)

Move memory contents from low address to high address (MOVUP)

CARTRIDGE PROGRAM SEARCH ROUTINE

A cartridge search routine is used to find a specified program or subprogram through the linked list in cartridge memory. The name of the desired program or subprogram is input to the routine, and a pointer to the header of the first program or subprogram with that name is returned. Since the search routine does not verify program type, main programs and subprograms should not be given the same name. Only the first occurrence of a name will be found.

The routine used to search cartridge memory for a program is the following.

Search the cartridge memory for a specified program/subprogram (CRTLST)

RANDOM INTEGER ROUTINE

The random integer routine generates a 16- and 8-bit pseudo-random integers. The 8-bit integer is returned in both binary and BCD form. The routine is the following.

Generate pseudo-random integer (RNDBYT)

SOUND GENERATION ROUTINE

The system provides a single sound routine that generates a short tone through the beeper. This routine is identified as follows.

Beeper routine (BEEP)

DEBUG MONITOR ENTRY ROUTINE

The Debug MONITOR is used to debug programs developed in assembly language. The monitor may be entered during execution of a machine-language program through a routine in system ROM. Complete information concerning the use of the Debug Monitor can be found in chapter 8 of the CC40 Editor/Assembler User's Guide.

The routine used to enter the Debug Monitor from a machine-language program is the following.

Enter the debug monitor (MONPEG)

POWER-UP, POWER-DOWN, AND BATTERY CHECK ROUTINES

System power-up and power-down operations are performed by software routines in the system ROM. The power-down routine can be used to turn the computer off. Transferring control to the power-up routine has the same effect as pressing the RESET button during program execution. The system also provides a routine for checking the condition of the batteries.

The three routines which control and check console power are as follows.

Check condition of batteries (BATCHK)
Power-down routine (POWDWN)
Power-up routine (POWUP)

I/O SUBSYSTEM

The I/O subsystem consists of a single subroutine called IOS that performs all communications on the HEX-BUS-tm. The subsystem makes extensive use of a data structure called a peripheral access block (PAB) to control communications with

peripheral devices.

Making a call through IOS to perform an I/O operation includes the following steps.

1. Building PAB that is apropriate for desired peripheral device operation.

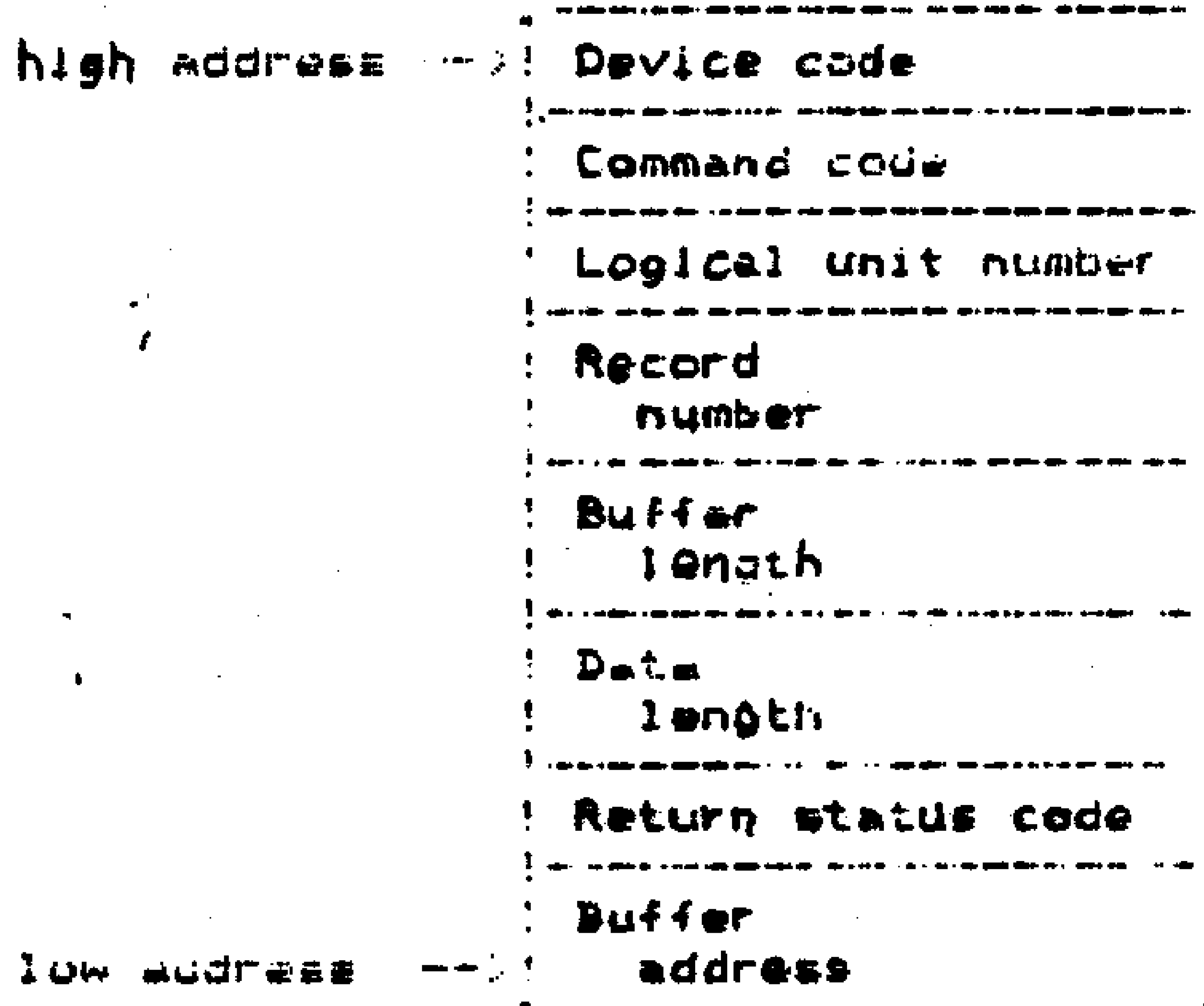
2. Placing the address of the PAB in the first two bytes of the floating point accumulator (FAC) with the most significant byte of the address in register >75 and the least significant byte in >76.

3. Calling the I/O subroutine (IOS).

4. Testing the status byte in the PAB for an error condition after the return from the call to IOS. The status byte is pointed to by the first two bytes of FAC.

PERIPHERAL ACCESS BLOCK (PAB)

The PAB contains the information needed by the I/O subsystem to communicate with a peripheral device. The following diagram shows the information which the PAB contains and the order in which the information must appear. Two-byte fields are stored with the least significant byte of the value at the highest address.



The **device code** is a number that is unique for each peripheral device on the bus. This code determines the device with which the I/O subsystem communicates.

The **command code** is a number that specifies the operation to be performed by the peripheral device. Commonly used command codes and their uses are described below.

The **logical unit number (LUNO)** is used by peripheral devices that permit access to device sub-units such as multiple open files on a single mass-storage device. A LUNO must be non-zero and must be uniquely assigned to a device sub-unit.

The **record number** is used by devices that support relative record files. Relative record files are those that permit individual records to be accessed in any order desired. The record number is ignored in accesses to sequential files. The first record of a file is record number zero.

The buffer length specifies the size of the I/O buffer that is used to store data to be sent to the peripheral or data that is received from the peripheral.

The data length specifies the actual number of bytes of data in the I/O buffer. The data is stored in reverse order from the highest buffer address through the lowest address.

The return status code is a number that is sent by the peripheral device to indicate the status of the attempted operation. If zero is returned, the requested operation has been completed without error. Any non-zero value is an error code that indicates the cause of the failure, such as a file not being open or a verify error occuring.

The buffer address is the address of the highest byte of the I/O buffer.

I/O CALLS

Communication with a peripheral device is performed with a sequence of calls to the I/O subsystem. First, the device is set up for HEX-BUS-tm communications with an "open" I/O call. Next, the device is accessed by I/O calls that "read" or "write" data or perform some other operation. When the operations are completed, device operations are completed and communications are terminated by a "close" I/O call.

The following table lists common operations and command codes used by IOS to communicate with peripheral devices.

Operation	Command Code
OPEN device or file	>00
CLOSE device or file	>01

DELETE OPEN FILE . . . . >02
READ data from device or file . . . . >03
WRITE data to device or file . . . . >04
RESTORE file position . . . . >05
DELETE FILE from device . . . . >06
RETURN STATUS information on device . . . . . >07
FORMAT mass storage medium . . . . >0D
READ CATALOG from device. . . . . >0E
SET OPTIONS for device. . . . . >0F
RESET all devices on the HEX-BUS . . . . . . >FF

In the descriptions of each I/O operation which follow, the requirements for building a PAB are listed. Whenever a PAB field is not listed in the description, the field is ignored during the particular I/O operation.

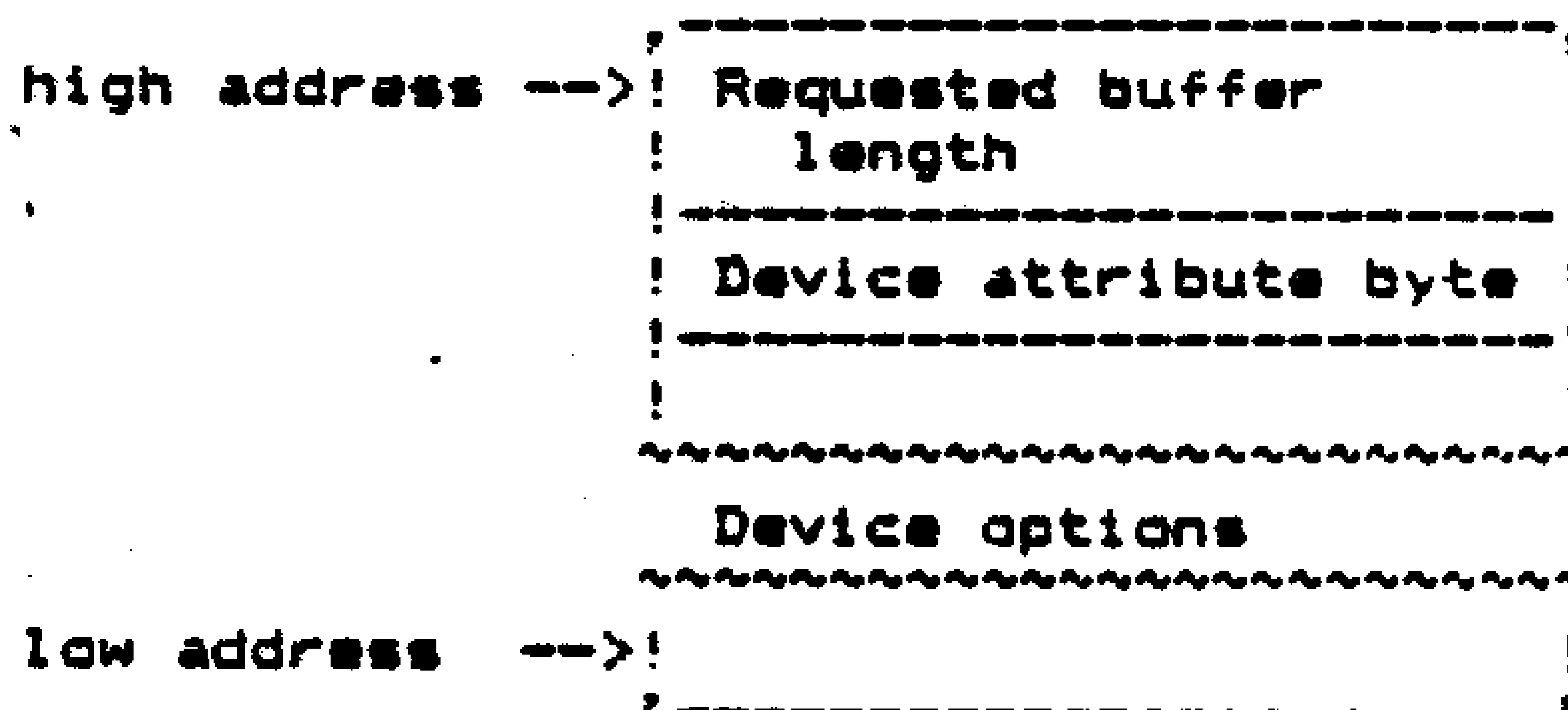
OPEN Command (>00)

The OPEN command is used to initialize communication with a HEX-BUS device. The device checks the requested access mode and buffer length and makes sure that the device is not already open. The PAB for an OPEN call should be built as shown in the following table.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>00
Logical unit number	Unique non-zero value to be associated with the device on file to be opened
Record number	>0000
Buffer length	Length of the "open" I/O buffer (must be at least four bytes)
Data length	Length of the data to be sent to the device in the open operation
Return status	>00 - A status value is returned by the peripheral device.

Buffer address	Pointer to the high end of the I/O buffer

The I/O buffer must contain the following information to be sent to the peripheral device during the open operation.



The peripheral compares the requested buffer length to its own capabilities. If it can support the requested length, it returns the requested length. If the requested length is zero, the peripheral returns the default buffer length specified in the device. If the requested length is a non-zero value that cannot be supported by the device, an error code is returned in the return status byte.

The device attribute byte contains flag bits that indicate the access mode and attributes of the device or file that is being opened. Two of the flags are ignored by peripheral devices and may be used as desired. The following table shows the definitions of the flags. Bit 0 is the least significant bit.

Bit(s)	Flag meaning
7-6	Access mode = 00 -> append mode = 01 -> input mode = 10 -> output mode = 11 -> update mode
5	File organization = 0 -> sequential organization = 1 -> relative organization
4	Record type = 0 -> variable length records = 1 -> fixed length records
3	File data type = 0 -> display type data = 1 -> internal type data
2	Device dependent use
1-0	Ingored by peripheral devices: available for use

The peripheral verifies that the requested access mode matches its own capabilities. If the access mode is not supported, an error is returned. The access modes are defined as follows.

Append mode specifies that data can only be written (or appended) to the end of a file. (The peripheral returns the number of the next record to be written in the returned data from the open operation.)

Input mode specifies that data can only be read from a device or file.

Output mode specifies that data can only be written to a device or file. Output begins with the first record of the file.

Update mode specifies that data can be both read from and written to the device or file.

Bit 2 of the device attributes is reserved for device-dependent usage. Software that does not use any special device dependent feature requiring this flag should set the bit to zero.

The device options field contains ASCII text that indicates the desired status of any device dependent characteristics. Examples of this type of information are the file name for a mass storage device, the carriage return and line feed settings for a printer, and the baud rate and parity settings for an RS-232 device. The device options field can remain empty if no device dependent options are involved in the open. However, the requested buffer length and the device attributes byte are always required. If these three bytes are not present for the open, an error is returned. Even when the device options field is empty, the open I/O buffer must be a least four bytes long because a successful open operation returns four bytes of information.

The following example shows a PAB and associated I/O buffer used to open an RS-232 device with a device code set to 20. The requested buffer length is 80 bytes. The device attributes are input mode, sequential organization, variable length records, and display type data. The device dependent options are a baud rate of 4800 (B=4800), even parity (P=E), and perform parity check (C=Y). The logical unit number for this example is 1.

```
                      PAB
high              .-----------.
address  ->!  >14         !  device code
              !-----------!
              !  >00        !  command code
              !-----------!
              !  >01        !  logical unit number
              !-----------!
              !  >0000      !  record number
              !-----------!
              !  >0011      !  open I/O buffer length
              !-----------!
              !  >0011      !  open data length
              !-----------!
              !  >00        !  return status
              !-----------!
low           !  buffer     !  pointer to open I/O buffer
address  ->!  address    !
              '-----------'
```

```
                          Open I/O buffer
high              .-------------------------------.
address  ->!  >0050                         !  requested buffer length
              !-------------------------------!
              !  >40                           !  device attributes
              !-------------------------------!
              !  >42 >3D >34 >38 >30 >30 >2C   !  B = 4 8 0 0 ,
              !-------------------------------!
              !  >50 >3D >45 >2C               !  P = E ,
low           !-------------------------------!
address  ->!  >43 >3D >59                   !  C = Y
              '-------------------------------'
```

When an open operation is successful (return status = 0), four bytes of information are returned in the open I/O buffer. The accepted I/O buffer length is in the highest addressed two bytes. The next two lower addressed bytes contain the record number to which the file is opened. The calling program must use this information to initialize the PAB for subsequent I/O operations.

The following table shows the return status codes that can be returned by an OPEN I/O call.

Return status	Meaning
>00	Successful open
>01	Device options error
>02	Error in device attributes byte
>03	File not found
>05	File or device already open
>06	Device related error
>09	File or device write protected
>0B	Directory full error
>0C	Buffer size error
>11	File organization (sequential/relative) incorrect or not supported
>13	Append access mode not supported
>14	Output access mode not supported
>15	Input access mode not supported
>16	Update access mode not supported
>17	File data type (display/internal) incorrect or not supported
>19	Low batteries in peripheral device
>1A	Uninitialized medium
>1B	Error detected in bus transfer
>20	Media full

CLOSE Command (>01)

The CLOSE command is used to terminate communication with a specific device. When a CLOSE command is received, the device can perform any necessary house-keeping tasks such as writing an end-of-file marker. The following PAB fields should be initialized as shown.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>01
Logical unit number	Unique value specified in the open operation
Data length	>0000 - No data is sent for a CLOSE command.
Return status	>00 - A status value is returned by the peripheral device.

A **CLOSE** command must be issued before another open command can specify the same file or device. An error also occurs when a **CLOSE** command specifies a file or device that is already closed (or has never been opened). The following table shows the return status codes that can occur during a CLOSE I/O call.

Return status	Meaning
>00	Successful close operation
>04	Device or file already closed (or never opened)
>06	Device related error
>08	Data or file too long error

DELETE OPEN FILE Command (>02)

Sometimes it is desirable to delete a file that is currently open: for example, it may be desirable to delete a temporary file immediately after reading its contents. In these cases the **DELETE OPEN FILE** command can be used in place of a **CLOSE** command followed by a **DELETE** command. The following PAB fields should be initialized as shown for a DELETE-OPEN-FILE I/O call.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>02
Logical unit number	Unique value specified in the open operation
Data length	>0000 - No data is sent for a delete-open-file command.
Return status	>00 - A status value is returned by the peripheral device.

The following return status codes can occur during a DELETE OPEN FILE I/O call.

Return status	Meaning
>00	Successful operation
>04	Device or file not currently open
>06	Device related error
>09	File or device write protected
>0D	Command not supported
>1C	File is delete protected

READ DATA Command (>03)

The READ DATA command is used to request a record of data from an open device or file. The input or update access mode must currently be selected or an error occurs. The PAB for a READ DATA I/O call is as follows.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>03
Logical unit number	Unique value specified in the open operation
Record number	Number of the record to be read
Buffer length	Accepted buffer length returned by the device during the open operation
Data length	>0000 - No data is sent for a READ data command.
Return status	>00 - A status value is returned by the peripheral device.
Buffer address	Address of the I/O buffer into which the data is read.

Upon completion of the I/O call, the calling program must check the return status code for any possible error. If the READ DATA command is successful, the I/O buffer contains the requested data in reverse order from the high end of the buffer to the low end. The data length field contains the number of bytes in the buffer. After each successful READ DATA I/O call, the calling program should increment the record number field of the PAB so that relative record files are properly supported. The following return status codes can occur during a READ DATA I/O call.

Return status	Meaning
>00	Successful read-data operation
>02	Attribute error
>04	Device or file not open
>06	Device related error
>07	End-Of-File (EOF) error
>0C	Buffer size error
>0D	Command not supported
>0F	File or device not open for read

WRITE DATA Command (>04)

The WRITE DATA command is used to send a data record to an open device or file. The output, update, or append access mode must be currently selected or an error occurs. The following PAB fields are initialized as shown for a WRITE DATA I/O call.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>04
Logical unit number	Unique value specified in the open operation
Record number	Number of the record to be written
Data length	Actual length of the data record to be written
Return status	>00 - A status value is returned by the peripheral device.
Buffer address	Address of the I/O buffer from which the data is written

Upon completion of the I/O call, the calling program must check the return status code for any possible error. If the WRITE DATA command is successful, the calling program should

increment the record number field of the PAB so that relative record files are properly supported. The following return status codes can occur during a WRITE DATA I/O call.

Return status	Meaning
>00	Successful write-data operation
>02	Attribute error
>04	Device or file not open
>06	Device related error
>08	Data or file too long error
>09	File or device write protected
>0C	Buffer size error
>0D	Command not supported
>0E	File or device not open for write

RESTORE command (>05)

The **RESTORE** command is used with mass storage devices to position an open file to its first record. The PAB fields in the following table are initialized as shown.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>05
Logical unit number	Unique value specified in the open operation
Data length	>0000 - No data is sent for a restore operation.
Return status	>00 - A status value is returned by the peripheral device.

The following return status codes can occur during a **restore** I/O call.

Return status	Meaning
>00	Successful restore operation
>04	Device or file not open
>06	Device related error
>0D	Command not supported

DELETE Command (>06)

The DELETE command is used to remove closed files from mass storage devices. The following PAB fields are initialized as shown in the table.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>06
Data length	Length of the data to be sent to the device
Return status	>00 - A status value is returned by the peripheral device.
Buffer address	Pointer to the high end of the I/O buffer

The I/O buffer contains the name of the file to be deleted. The file name is specified in the same manner as the options are in the open operation: that is, it is stored as ASCII text in reverse order from the high end of the buffer toward the low end. The following return status codes may occur.

Return status	Meaning
>00	Successful delete operation
>03	File not found
>05	File is open
>06	Device related error
>09	File or device write protected
>0D	Command not supported
>1C	File is delete protected

RETURN STATUS Command (>07)

The RETURN STATUS command is used to read device status information. A variety of status information is returned to the calling program in the I/O buffer in the form of a single byte containing defined flag bits. If sufficient I/O buffer length is provided, some peripheral devices return more status information than the standard one byte. The PAB is initialized as shown in the following table. The record number is ignored.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>07
Logical unit number	Unique value specified in the open operation, or >00 for general device status
Buffer length	Length of I/O buffer into which status is read (must be greater than or equal to one)
Data length	>0000 - No data is sent for a RETURN STATUS operation.
Return status	>00 - A status value is returned by the peripheral device.
Buffer address	Pointer to the high end of the I/O buffer

When the logical unit number is the unique non-zero value associated with an open file or device, the status information for that file or device is returned. When the logical unit number is specified as zero, general device status is returned. The device is not required to be open when LUNO zero is used; however, mass storage devices that only support one open file will return file status. The flag bits in the standard status byte are shown in the following table. Bit 0 is the least significant bit.

Bit(s)	Flag meaning
7	End-of-file status
	= 0 -> not at end of file
	= 1 -> at end of file
6	Random access support
	= 0 -> file or device does not support random access
	= 1 -> file or device does support random access
5	File protection
	= 0 -> file is not protected
	= 1 -> file is protected
4	Open status
	= 0 -> file or device is not open
	= 1 -> file or device is open
3-2	File or device type
	= 00 -> display type file or device
	= 01 -> internal type file or device
	= 10 -> data communications type device
	= 11 -> undefined
1-0	Supported access modes
	= 00 -> undefined
	= 01 -> file or device only supports read access
	= 10 -> file or device only supports write access
	= 11 -> file or device supports read and write access

Some devices can return further device dependent information if sufficient I/O buffer space is provided. For example, a data-communications device (HEX-BUS-tm RS232 or MODEM, for example) can return a two-byte value indicating the number of bytes

remaining in its output buffer. Device-dependent status information that can be obtained through a RETURN STATUS I/O call is described in the manual for each peripheral device.

The following can result from a RETURN STATUS operation.

Return status	Meaning
>00	Successful read-status operation
>04	Device or file not open
>12	Buffer size error

FORMAT Medium Command (>0D)

The FORMAT medium command instructs a mass storage device to initialize the currently installed storage medium. The following table shows the PAB fields for this I/O call. The logical unit number and record number can be any values.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>0D
Buffer length	Length of I/O buffer
Data length	Length of data to be sent to device
Return status	>00 - A status value is returned by the peri. device.
Buffer address	Pointer to the high end of the I/O buffer

Each type of mass storage device has a default procedure for formatting its storage medium. Some devices also provide the capability to specify formatting options. These options are described in the manual for the device. If the data length in

the PAB is zero, the device formats the medium using its default procedure. If the data length is non-zero, the data contained in the I/O buffer specifies desired formatting options. The following return status values can occur.

Return status	Meaning
>00	Successful format-medium operation
>05	Device or file is open
>06	Device related error
>09	File or device write protected
>0D	Command not supported

READ CATALOG Command (>0E)

The READ CATALOG command instructs a mass storage device to return directory information about a particular file. The PAB for a READ CATALOG I/O call is initialized as follows. The logical unit number can be any value.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>0E
Record number	File position in directory
Buffer length	>0012
Data length	>0000
Return status	>00 - A status value is returned by the peripheral device.
Buffer address	Pointer to the high end of the I/O buffer

The file number is the position of the file in the directory. The first file is number zero. A complete directory can be cataloged by starting with file number zero and incrementing the file number after each successful read-catalog I/O call. The end of the directory has been reached when a "File not found" error (>03) is returned. A successful READ CATALOG command returns the following 18 bytes of information in the I/O buffer. They are stored in reverse order from the high end of the buffer toward the low end.

File number	1 byte
File name	12 bytes
Maximum Record length	2 bytes
Number of records	2 bytes
Device dependent flags	1 byte

The following return status codes may occur.

Return status	Meaning
>00	Successful read-catalog operation
>03	File not found
>05	Device or file is open
>06	Device related error
>0C	Buffer length error
>0D	Command not supported

SET OPTIONS Command (>0F)

The SET OPTIONS command can be used to send device options as defined in the OPEN command without closing and re-opening the device. The ASCII text that specifies the desired status of the device dependent options is stored in reverse order starting at the high end of the I/O buffer. The PAB fields are initialized

as follows. Since the record number is ignored, it can be any value.

PAB field	Contents
Device code	Unique code number for the desired device
Command code	>0F
Logical unit number	Unique value specified in the open operation
Data length	Length of the data to be sent to the device
Return status	>00 - A status value is returned by the peripheral device.
Buffer address	Pointer to the high end of the I/O buffer

The following return status codes can occur during a SET-OPTIONS I/O call.

Return status	Meaning
>00	Successful set-device-options operation
>01	Error in device options
>04	Device or file not open
>06	Device related error
>0D	Command not supported

RESET Command (>FF)

The RESET command sets all devices on the HEX-BUS-tm to their power up status. This command closes all open files and devices on the bus. The PAB fields for this command are as shown below. All other fields are ignored.

PAB field	Contents
Device code	>00
Command code	>FF
Data length	>0000

The return status code following a reset-bus command should be ignored.

USE OF THE IOS FROM BASIC

The I/O subsystem may be called directly from BASIC as well as from machine language.

In addition to the standard I/O commands of statements (INPUT and PRINT, for example), BASIC includes a built-in subprogram (IO) that can be called by a BASIC program or subprogram to access a device on the HEX-BUS. The syntax of the call from BASIC is as follows.

```
CALL IO(device.command[,status-variable])
or
CALL IO(string-variable[.status-variable])
```

where

device is a numeric expression whose value is the unique device code associated with a peripheral device.

command is a numeric expression whose value is the command code that specifies the operation to be performed.

string-variable is a simple string variable that contains from 1 to 12 characters that represent all or part of a peripheral access block.

status-variable is a numeric variable in which the return status from the I/O operation is stored. The brackets [ ] around ".status-variable" indicate that it is optional.

The IO subprogram begins by setting the contents of the PAB to zero and then storing >0080 as the buffer length and >00FF as the buffer address. If the first syntax is used, the result of the device expression is placed in the device code field of the PAB, and the result of the command expression is placed in the command code field. If the second syntax is used, the value of the simple string variable is copied into the PAB starting at the device code field. The minimum contents of the string are, therefore, the device code and command code. If the string contains more than 12 characters, the extra characters are ignored.

Once the PAB is initialized, the IO subprogram performs the I/O call. If the second syntax is used, the contents of the PAB following the I/O operation are assigned to the string variable. Thus, the string variable contains 12 characters following the operation regardless of the number of characters prior to the operation.

After completion of the I/O operation, IO checks for the status-variable. If it is present, the return status in the PAB is assigned to the numeric variable. If the status-variable is not present and an error occurred, the error is reported to the BASIC error handler.

PAB field	Contents
Device code	>00
Command code	>FF
Data length	>0000

The return status code following a reset-bus command should be ignored.

USE OF THE IOS FROM BASIC

The I/O subsystem may be called directly from BASIC as well as from machine language.

In addition to the standard I/O commands of statements (INPUT and PRINT, for example), BASIC includes a built-in subprogram (IO) that can be called by a BASIC program or subprogram to access a device on the HEX-BUS. The syntax of the call from BASIC is as follows.

```
CALL IO(device.command[,status-variable])
or
CALL IO(string-variable[,status-variable])
```

where

device is a numeric expression whose value is the unique device code associated with a peripheral device.

command is a numeric expression whose value is the command code that specifies the operation to be performed.

string-variable is a simple string variable that contains from 1 to 12 characters that represent all or part of a peripheral access block.

status-variable is a numeric variable in which the return status from the I/O operation is stored. The brackets [ ] around ".status-variable" indicate that it is optional.

The IO subprogram begins by setting the contents of the PAB to zero and then storing >0080 as the buffer length and >00FF as the buffer address. If the first syntax is used, the result of the device expression is placed in the device code field of the PAB, and the result of the command expression is placed in the command code field. If the second syntax is used, the value of the simple string variable is copied into the PAB starting at the device code field. The minimum contents of the string are, therefore, the device code and command code. If the string contains more than 12 characters, the extra characters are ignored.

Once the PAB is initialized, the IO subprogram performs the I/O call. If the second syntax is used, the contents of the PAB following the I/O operation are assigned to the string variable. Thus, the string variable contains 12 characters following the operation regardless of the number of characters prior to the operation.

After completion of the I/O operation, IO checks for the status-variable. If it is present, the return status in the PAB is assigned to the numeric variable. If the status-variable is not present and an error occurred, the error is reported to the BASIC error handler.

CHAPTER 3
RUNNING PROGRAMS IN THE BASIC ENVIRONMENT

BASIC OPERATING ENVIRONMENT

System subroutines, discussed in the last chapter, are parts of the interpreter and system code which are useful within programs and subprograms developed in assembly language. In this sense, the system in ROM in the CC-40 "supports" programs developed by assembly language.

In a different sense, programs developed in assembly language support BASIC programs. The prevalent use of subprograms written in assembly language is to perform various functions within a BASIC program during the time the BASIC program is running. Integer arithmetic operations, for example, can be performed in machine-language programs much faster than they can be through the floating point operations of BASIC. Values to be used in integer arithmetic functions can be input to a BASIC program and transferred to a machine-languge subprogram; the integer value of the function obtained by machine-language calculation can be ruturned to the BASIC program.

Because BASIC is a machine-language program using CC-40 hardware resources, writing a machine language subprogram through assembly language requires a knowledge of the resources used by BASIC. For the source and destinaton of values it processes, a machine-language subprogram often uses memory which BASIC also uses for tasks such as passing values to and from a BASIC program. Equally important, a machine-language program must avoid the "destructive" use of memory used by BASIC: the machine-

language subprogram must store values and be executed in areas of memory that are not in use by BASIC.

Designing machine-language programs to be executed while BASIC is running, then, requires a knowledge of the BASIC operating environment. Registers and RAM must be carefully accounted for in the assembly-language development process.

BASIC REGISTERS

The general-purpose register file of the CC-40 is segmented by BASIC into five distinct areas, as shown in the following memory map.

Area	Address
A and B registers	>00->01
Processor stack	>02 >39
Statement level temporary area	>3A >4A
Reserved pointers and flags	>4B >57
Function level temporary area	>58 >7F

The A and B registers are the most heavily used registers in assembly language programming, because they are used in the most versatile and economical instructions. To permit frequent use of these registers by programs developed in assembly language, BASIC does not store any values in them when a machine-language subprogram can be called.

The processor stack, on the other hand, is in use at all times by both BASIC and machine-language programs. Every CALL or TRAP instruction requires two bytes of this stack, every paged call (CALPAG, CTERP) requires five bytes, and many system routines push and pop data onto and off of the stack while execution is under their control.

The statement level temporary area is used during the interpretation of BASIC program statements, but they are free for use by machine-language programs after statement interpretation. After a BASIC CALL statement to a machine-language subprogram is interpreted and put into effect, the 17 registers in the statement temporary area are always available for subprogram use. If the machine-language program, however, calls routines in system ROM, these registers may be used. They are used by the data entry routines BUFIN and LINEIN and by the cartridge search routine CRTLST. In addition, system ROM routines such as OPEN, CLSTMP, CLSALL, and a memory clean-up routine called TRSHDY use this area extensively.

The reserved pointers and flags are organized as follows.

Item	Address
System flags	>4B, >4C
Current program character	>4D
BASIC program pointer	>4E, >4F
Parse stack pointer	>50, >51
Display start offset	>52
Cursor position	>53
Free space pointer	>54, >55
Floating point stack pointer	>56, >57

As shown in the memory map, this block of registers contains execution-control information and must remain unchanged throughout the execution of a machine-language subprogram.

The function level temporary area is used by almost all system software routines. Use of these registers,therefore, must be carefully planned when system ROM routines are being called.

BASIC RAM

BASIC segments the system RAM into eight distinct areas. The lowest addressed area (system reserved area) is fixed in memory from address >800 to address >991. Each of the seven other areas is delimited at the low end by the highest byte used in the previous area and at the high end by a two byte pointer to the highest byte in this area. Addresses of highest-byte pointers are shown in brackets in the following diagram.

```
,-------------------------------,
! System reserved area          !
! >800 through >991             !
!-------------------------------!
! Machine language subprograms  !
! [>990,>991]                   !
!-------------------------------!
! User assigned string table    !
! [>8E8,>8E9]                   !
!-------------------------------!
! Variable name table           !
! [>8EA,>8EB]                   !
!-------------------------------!
! Floating point stack          !
! [>56,>57]                     !
!-------------------------------!
! Free memory space             !
! [>54,>55]                     !
!-------------------------------!
! Dynamic memory area           !
! [>8EC,>8ED]                   !
!-------------------------------!
! Main program area             !
! [>800,>801]                   !
'-------------------------------'
```

System Reserved Area

The system reserved area is a fixed block of memory between addresses >800 and >991. This area stores a wide variety of system control information, and it must be used with great care. The following table lists the usage of this area by address. Locations within this area which are generally available to machine-language programs are discussed later in this chapter.

Locations	System Usage
>800,>801	Pointer to highest RAM address (which is also the highest address in the main program area)
>802 to >804	System power up information (used to determine necessary level of initialization during power-up)
>805 to >80F	Reserved
>812 to >827	Branch vectors for TRAPs 4 to 11
>828 to >82B	Pointers to main program and subprogram headers when running a BASIC program in a cartridge
>82C,>82D	Line number increment when command level is in auto-line-number mode
>82E,>82F	Pointer to linked list of BASIC subprogram headers in system RAM during BASIC program execution
>830	Auto-power-down status, zero indicates APD in effect, non-zero indicates APD not in effect
>831	Used by system to indicate whether a RAM cartridge has been appended to the system RAM
>832,>833	Used to maintain position of pending output to the display in BASIC
>834 to >839	Used to maintain important information between successive scans of the keyboard
>83A to >83D	Used to maintain current display indicator status

Locations	System Usage
>83E to >88D	Used by data entry routines as primary 80 character input buffer
>88E to >8A5	Used by I/O subsystem
>8A6,>8A7	Used by psuedo-random integer generator (RNDBYT)
>8A8,>8A9	Pointer to linked list of headers in the machine language subprogram area
>8AA to >8AF	Used to maintain current DATA context while executing a BASIC program
>8B0,>8B1	Pointer to the current program line while editing a BASIC program
>8B2,>8B3	Line number at which to stop when listing a BASIC program
>8B4 to >8C8	General temporary area used by BASIC mostly when converting floating point numbers to their string representations with the CNS routine
>8C9,>8CA	Pointer to the next BASIC program line following the one currently being executed
>8CB to >8DA	Floating point numbers used by the floating point random number generator (RND)
>8DB,>8DC	Pointer to the first program header in the cartridge
>8DD,>8DE	Pointer to the current BASIC statement being executed
>8DF,>8E0	Pointer to the current function (FN) key table associated with the non-user-assignable keys
>8E1,>8E2	Pointer to the symbol table for the currently executing BASIC program or subprogram
>8E3	Currently selected language indicator
>8E4,>8E5	Pointer to the imperative symbol table which contains those symbols created at the command level rather than during program execution
>8E6,>8E7	Pointer to first (highest addressed) free block in linked list of free space
>8E8,>8E9	Pointer to highest address in the user assigned string table

Locations	System Usage
>8EA,>8EB	Pointer to highest address in the variable name table which is also the base address of the floating point stack
>8EC,>8ED	Pointer to the highest address in the dynamic memory area
>8EE,>8EF	Pointer to the first entry in the linked list of peripheral access blocks
>8F0,>8F1	Reserved
>8F2,>8F3	Pointer to the header of the currently executing BASIC program or subprogram
>8F4 to >8F9	Used by the debug monitor to handle machine language breakpoints
>8FA	Pointer to the end of the processor stack in the register file
>8FB to >8FF	Used by BASIC to maintain current status of ON BREAK, ON WARNING, ON ERROR, and PAUSE ALL statements
>900 to >977	Used by BASIC line compression algorithm to convert program lines from ASCII text into internal BASIC format; Also used at the command level as the playback buffer
>978 to >98F	Used by the expression evaluation algorithm (parser) as an operator precedence stack
>990,>991	Pointer to highest address in the machine language subprogram area

Machine Language Subprogram Area

The machine language subprogram area contains subprograms loaded into memory by the LOAD subprogram in system ROM. Each time the LOAD subprogram is called, the user assigned string table, the variable name table, and the floating point stack are moved up in memory, and the subprogram area is expanded to accomodate the new subprogram(s).

The system maintains a pointer [>990,>991] that indicates the highest address in this area. This pointer is also used to delimit the low end of the user assigned string table. Another pointer [>8A8,>8A9] points to the first header in the linked list of subprograms in this area. When new subprograms are loaded into the area, the headers of the subprograms are added to the end of the existing previous list. The low end of this area is always at address >992.

User Assigned String Table

The user assigned string table is a table of strings assigned to the ten number keys (0-9). When this table is initialized, it consists of ten zeros, each representing a null string. When a string is assigned to a particular number key, that string replaces the corresponding null string (or any previously assigned string) in the table.

The system maintains a pointer [>8E8,>8E9] that indicates the highest address used by the table. This address is also the address of the length byte of the string assigned to the "0" key. The next lower string in the table belongs to the "1" key, and so on. The string corresponding to a particular key is found by using the length bytes (plus 1) to step through the table from high to low until the string in the correct position is found. The low end of this table is delimited by the high end pointer [>990,>991] of the machine language subprogram area.

Variable Name Table

When a BASIC statement is entered, the variable names are "tokenized"; that is, each variable is assigned a one byte value

which represents the variable name in the "internal form" used by BASIC. The string representation of the name is then placed in the variable name table. The token value assigned to a name correponds to the position of the name in the table biased by >20. Thereafter, each time the same variable name is used, it is represented by this same token value.

The table also uses two bytes of overhead. At the low end of the table is a zero, representing a null string. The purpose of the null string is to terminate the search used for finding variable names. At the high end of the table is a byte which contains the next available token value to be assigned to the next new variable name. Within the table, each variable name is the string representation of the name (length byte above ASCII text) with the most significant bit of the last character of the name set to ONE.

The system maintains a pointer [>8EA,>8EB] that indicates the highest address used by the table. This address is the address of the next-available token byte. The search algorithm uses the length bytes to step through the table until the correct variable name is found. The low end of the table is delimited by the high end pointer [>8E8,>8E9] of the user assigned string table.

Floating Point Stack

The floating point stack, in addition to its being used in floating point routines as described in chapter 2, is used by the BASIC interpreter as an execution-control stack. The interpreter uses the stack to save a variety of data structures including

variable information entries, FOR-NEXT loop entries, GOSUB-RETURN entries, subprogram CALL entries, error codes, and BASIC breakpoint entries. Each of these structures requires one or more eight byte stack entries. (Variable information entries are discussed later in this chapter in connection with subprogram parameter passing). The other structures allow the interpreter to properly control the flow of a running BASIC program.

The system maintains several pointers to the floating point stack including the floating point base address [>8EA,>8EB], the floating point stack pointer [>56,>57], and the free space pointer [>54,>55]. The floating point base address pointer is the same as the high end pointer for the variable name table.

Free Memory Space

Free memory space is present between the floating point stack and the dynamic memory area. The floating point stack extends into the free memory space as necessary, and dynamic memory management allocates space "from above" into this area as a last resort when it fails to find a sufficiently large block in the linked list of free memory blocks. If a stack push would ever cause the value of the stack pointer to exceed the value of the free space pointer, or if dynamic memory management cannot find a large enough block including the free memory space, the system reports a "Memory full" error.

The free memory space pointer is maintained in registers >54,>55. This pointer indicates the highest byte that can be used by the floating point stack and the highest byte of the last available free block for the dynamic memory manager. The low end

of this last free block is delimited by the stack pointer [>56,>57].

Dynamic Memory Area

The dynamic memory area is used by the BASIC interpreter to store run-time data structures, including the following.

- o values of string variables
- o temporary strings used during evaluation of string expressions
- o peripheral access blocks and associated I/O buffers
- o subprogram symbol tables

When a block of memory is needed, the interpreter calls the appropriate memory allocation routine. The allocation routine searches a linked list of free memory blocks until it finds a sufficiently large area (or fails to do so and reports "Memory full"). When a block is no longer needed, the interpreter calls the appropriate deallocation routine, and the deallocation routine links the released block into the linked list of free memory blocks.

The system maintains a pointer [>8EC,>8ED] to the highest addressed byte in the dynamic memory area and a pointer [>8E6,>8E7] to the linked list of free memory blocks. The free space pointer [>54,>55] indicates the last available free block, and the floating point stack pointer [>56,>57] delimits the low end of this area.

Main Program Area

The system permits one main program in memory at a time. That program resides in the main program area. The low end of the main program is delimited by the dynamic memory base address pointer [>8EC,>8ED], and the high end is the last available byte of memory, always pointed to by the high RAM pointer [>800,>801]. The high end is also the location of the main program header.

MEMORY USAGE AND SYSTEM REENTRY

Programs and subprograms developed in assembly language can be classified into four groups depending upon the extent to which they use register file and the system reserved area of memory. Each level of memory usage requires a distinct method of returning execution control to system software.

The four levels are numbered from zero to three: level zero corresponds to the least extensive memory usage, and level three corresponds to the most extensive memory usage.

LEVEL ZERO MEMORY USAGE

Level zero usage is for programs and subprograms designed to run entirely within the context of the BASIC operating environment. Although main programs can be designed to run at this level, the most common use for this level is an machine language subprogram designed to be called by a BASIC program which is currently running.

Because level zero programs must return control to a BASIC program, they may use only those elements of the BASIC operating environment currently unused by BASIC in running its program. Subprogram calls from BASIC are at the statement level of

execution, and therefore level zero subprograms may use the entire statement level temporary area. Also, they may use the function level temporary area. If system subroutines are to be called, however, use of these two areas must be carefully designed around register usage in the called routines.

Level zero subprograms can use the dynamic memory management subsystem to allocate and deallocate blocks of memory for their own uses. They can also use the FPPUSH, FPPOP, and POPARG routines to manipulate entries on the floating point stack.

Traps 4 through 11 transfer control through a branch table that is located in the system reserved area. Two of these traps (numbers 9 and 10) are designed to be used by level zero subprograms. TRAP 9 enters the table at location >81F, and TRAP 10 enters the table at location >822. A level zero subprogram can direct these TRAP instructions to any desired location by storing appropriate branch instructions at these locations.

Level zero subprograms return control to the calling BASIC program by executing the machine-language code for a RETS instruction. Level zero main programs can return control to the system by branching to the RETSYS entry point.

LEVEL ONE MEMORY USAGE

Level one usage is for programs and subprograms that are designed to run outside the restrictions of the BASIC operating environment but which maintain most of the memory structures defined above. Level one programs and subprograms may use the memory areas described in level zero. They may also use the floating point stack and the dynamic memory area through the

routines in system ROM. If they do not use these areas through ROM routines, all RAM between the variable name table and the main program area is free for program use.

In addition, the following register file and system RAM locations can be used provided that the routines associated with them are not also being used.

Registers	System Usage
>4D - >51	Used by subprogram parameter passing routines (GETNUM, GETSTR, GETADR, and GETCHR described later in this section)
>52 - >53	Used by the display control routines and the data entry routines (CLRDSP, DSPBUF, DSPCHR, SETCRS, BUFIN, LINEIN)
>54 - >57	Used by dynamic memory management subsystem and floating point subsystem (see previous chapter)

System RAM	System Usage
>828 to >82F	Available because BASIC is not running
>832,>833	" " " " " "
>83E to >88D	Used by data entry routines (BUFIN, LINEIN, CLRINP, MVINPB)
>8A6,>8A7	Used by psuedo-random integer generator (RNDBYT)
>8AA to >8B3	Available because BASIC is not running
>8B4 to >8C8	" " " " " "
>8C9,>8CA	Available because BASIC is not running
>8CB to >8DA	Used by floating point random number generator (RND)
>8DD,>8DE	Available because BASIC is not running
>8DF,>8E0	Used by data entry routines in connection with non-user-assignable function (FN) keys (BUFIN, LINEIN)

System RAM	System Usage
>8E1,>8E2	Available because BASIC is not running
>8E4,>8E5	" " " " " "
>8E6,>8E7	Used by dynamic memory management subsystem (MNEW, MDISP, STGNEW, STGDSP)
>8FA to >8FF	Available because BASIC is not running
>900 to >977	Used by the machine language Debug Monitor
>978 to >98F	Available because BASIC is not running

Level one programs, however, must maintain the main program area, the variable name table, the user assigned string table, the machine language subprogram area, and all associated control pointers.

Unlike level zero subprograms which are called from a running BASIC program, level one subprograms are designed to be called only from the command level of the system. A level one subprogram should therefore determine whether or not it has been called from a BASIC program by checking the "program running" flag, bit 5 of flag register >4B. If a BASIC program is running, the level one program should return to BASIC through the BASIC error handler to prevent memory usage other than that used in level zero to occur.

Often level one programs are implemented as a subprograms rather than as a main programs so that they can be kept in memory while other programs and subprograms are loaded and run. Examples of this type of program are the cartridge initialization (CARTINIT) and cartridge loading (CARTLOAD) subprograms.

Level one main programs are of two types, those designed to run repeatedly in the main program area of system RAM and those designed to run in the cartridge memory without destroying whatever main program may currently be in the system RAM.

All level one programs and subprograms return control to the system by branching to the RETSYS routine in system ROM. This routine reinitializes the floating point stack, dynamic memory management, and the associated control pointers.

LEVEL TWO MEMORY USAGE

Level two usage is for programs and subprograms that require more of the system RAM than level one allows. These programs may use areas reserved in lower levels of usage for the main program and variable name table. These programs can use all of the RAM memory between the assigned string area and the high end of system RAM. However, the pointer to the assigned string area [>8E8,>8E9] and the high RAM pointer [>800,>801] must be maintained. In addition to the memory usage described in levels zero and one, a level two program may alter the floating point base address pointer [>8EA,>8EB] and the dynamic memory base address pointer [>8EC,>8ED]. If the floating point stack and dynamic memory management are not used through system ROM routines, the associated RAM locations can be used by level two programs.

Because programs and subprograms designed for level 2 usage destroy the main program area and variable name table, they must return control to the system through the RETNEW routine in ROM. This routine initializes the main program area and variable name

table for a null program. It also initializes the floating point stack, the dynamic memory management, and associated control pointers. Because RETNEW initializes the main program area, a level two main program running in the main program area can only be run once before it must be reloaded. For this reason, most level two main programs are run in the cartridge memory.

LEVEL THREE MEMORY USAGE

Level three consists of programs and subprograms designed to use all RAM memory above the system reserved area (which ends at address >991). In addition to the memory usage for all the lower levels, level three programs may use the following system RAM locations provided that the associated system ROM routines are not being used.

Locations	Associated system usage
>8A8,>8A9	Pointer to linked list of headers in the machine language subprogram area
>8E8,>8E9	Pointer to highest address in the user assigned string table
>8F4 through >8F9	Used by the debug monitor to handle machine language breakpoints
>990,>991	Pointer to highest address in the machine language subprogram area

Since level three programs and subprograms destroy all of the memory structures used by BASIC, they must return control to the system by branching to the RETINT routine. This routine performs a complete initialization of the system, including display of the "System initialized" message. If a level three

program or subprogram is run in the system RAM, it is destroyed by the initialization procedure. These programs, therefore, are usually run in the cartridge memory.

TRAP VECTOR MEMORY USAGE

Twenty four bytes of RAM in the system reserved area may also be used by machine-language programs. These locations are most useful in providing for specialized uses for TRAP instructions (single-byte call instructions with implied addressing).

The system is initialized so that TRAP instructions 4 through 11 transfer control to a branch table copied from ROM into RAM at addresses >810 through >827. Traps 4 through 7 are used by a BASIC line-compression routine. Trap 8 is used throughout the system to transfer control to the BASIC error handler. Traps 9 and 10 are intended specifically for use by machine language programs (as described above for level zero memory usage). Trap 11 is used by the Debug Monitor in executing breakpoints in machine-language programs.

Each of these traps can be redirected by replacing the current branch vector with an a branch to the desired location. However, the following restrictions apply.

1. Level zero, one, and two programs and subprograms <u>must</u> save the current branch vectors for traps 4 through 8 and trap 11 before replacing them. These vectors must be restored before returning control to the system through a RETS instruction or the RETSYS or RETNEW routines. Level three programs and subprograms do not have to save the vectors because the RETINT routine completely initializes the system (including these vectors).

2. Many system subroutines use trap 8 to transfer control to the error handler. Either the corresponding vector must be used to redirect trap 8 to a machine-language error handling routine, or these system subroutines must not be used.

3. If trap 11 is redirected, machine-language breakpoints and the single step command will not function in the Debug Monitor.

SUBPROGRAM PARAMETER PASSING

Most subprograms require some sort of information to be provided to them for processing. Normally, this information is passed in the form of a list of arguments included in the CALL commands or statements from BASIC which transfer control to them. Subprograms must check the syntax of argument lists and must take the necessary steps to access each of the arguments. This process is called parameter passing.

Subprograms are executed with a CALL statement with the following syntax.

```
CALL subprogram_name(argument_list)
```

The argument list is optional. If no argument list is required, the parentheses are omitted. If the argument list is required, the subprogram defines the number, type, and order of the arguments that must be present. An individual argument can be any of the following depending upon the requirements of the subprogram.

1. a numeric constant or quoted string constant
2. a simple (single-element) numeric or string variable
3. a numeric or string expression
4. an element of a numeric or string array
5. an entire numeric or string array

As a subprogram accesses an argument list, it must verify that proper syntax has been used. Syntax checking consists of verifying the following.

1. That the argument list begins with a left parenthesis
2. That the list includes the correct number of arguments
3. That the arguments are separated by commas
4. That the argument list is terminated by a right parenthesis
5. That the CALL statement is terminated by an end of line

Tokens are the "internal form" of punctuation symbols used in BASIC commands and statements. Four token values, shown in the following table, are important in syntax checking. During parameter passing, a subprogram uses the compare instruction (CMP) to test a current character register (>4D) for the appropriate token values.

Character	Token
Left parenthesis	>C0
Comma	>AD
Right parenthesis	>AF
End-of-statement	>00

SYNTAX CHECKING AND PARMETER PASSING

Steps used in a subprogram for syntax checking and parameter passing are as follows.

1. Checking for a left parenthesis
2. Advancing the program pointer
3. Calling an appropriate argument access routine
4. If this argument is not the last one
   - 4a. Checking for a comma
   - 4b. Looping to step 2.
5. Checking for a right parenthesis
6. Advancing the program pointer
7. Checking for an end-of-statement token

Checking for a Left Parenthesis

When a CALL statement transfers control to the subprogram, the current character register (>4D) contains the character following the subprogram name. The BASIC program pointer [>4E,>4F] points to the character following the current character. If the subprogram is not designed to be called with an argument, the current character register can be checked for an end-of-statement token (>00). If the end-of-statement token is present, the subprogram can complete its processing and return control to the calling program. If the subprogram uses an argument list, the current character register is checked for a left parenthesis (>C0). If the left parenthesis is present, the subprogram continues with the parameter passing process. If the required characters are not present, the subprogram reports an

"Illegal syntax" error by loading the value >01 into the A register and executing a TRAP 8.

Advancing the Program Pointer

The first step in accessing any individual argument is to get the first character of the argument into the current character register and advance the BASIC program pointer. The system provides a TRAP routine (GETCHR) specifically for this purpose. This routine loads the next program character into the A register, advances the program pointer, copies the character from the A register to the current character register, and returns to the subprogram. (When control returns to the subprogram, the processor status byte is set according to the current program character.)

Calling the Appropriate Argument-Accessing Routine

The system provides four routines that are used to access individual arguments. These routines are the following.

```
GETNUM - Get the value of a numeric argument
GETSTR - get the value of a string argument
GETADR - Get the address of an argument
SYM    - Find a symbol in the current symbol table
```

These routines return the argument or information about the argument in a specified set of registers. When they return control to the subprogram, the current character register contains the character following the argument and the program pointer is advanced appropriately. If the argument is not what the subprogram requires, the subprogram should report a "Bad

argument" error (>1D) through a TRAP 8 instruction.

Getting Further Arguments

For further arguments in a list, the subprogram must check for the presence of a comma. If a comma is present, the current character register contains the comma token (>AD). If the comma is not present and the subprogram requires a further argument, the subprogram should report an "Illegal syntax" error (>01) to the error handler. If the comma is present, the subprogram proceeds by repeating steps 2 and 3 again. In other words the subprogram repeatedly checks for a comma token, advances the program pointer, and calls the appropriate argument accessing routine until the last argument has been accessed.

Checking for the Right Parenthesis

After getting the last "legal" argument, the subprogram should check for a right parenthesis rather than a comma. Absence of the right parenthesis can then be reported as an "Illegal syntax" error (>01) to the error handler.

Advancing the Program Pointer

In order to check for the end-of-statement token, the subprogram must get the next program character and advance the program pointer. The GETCHR routine is used to advance the pointer.

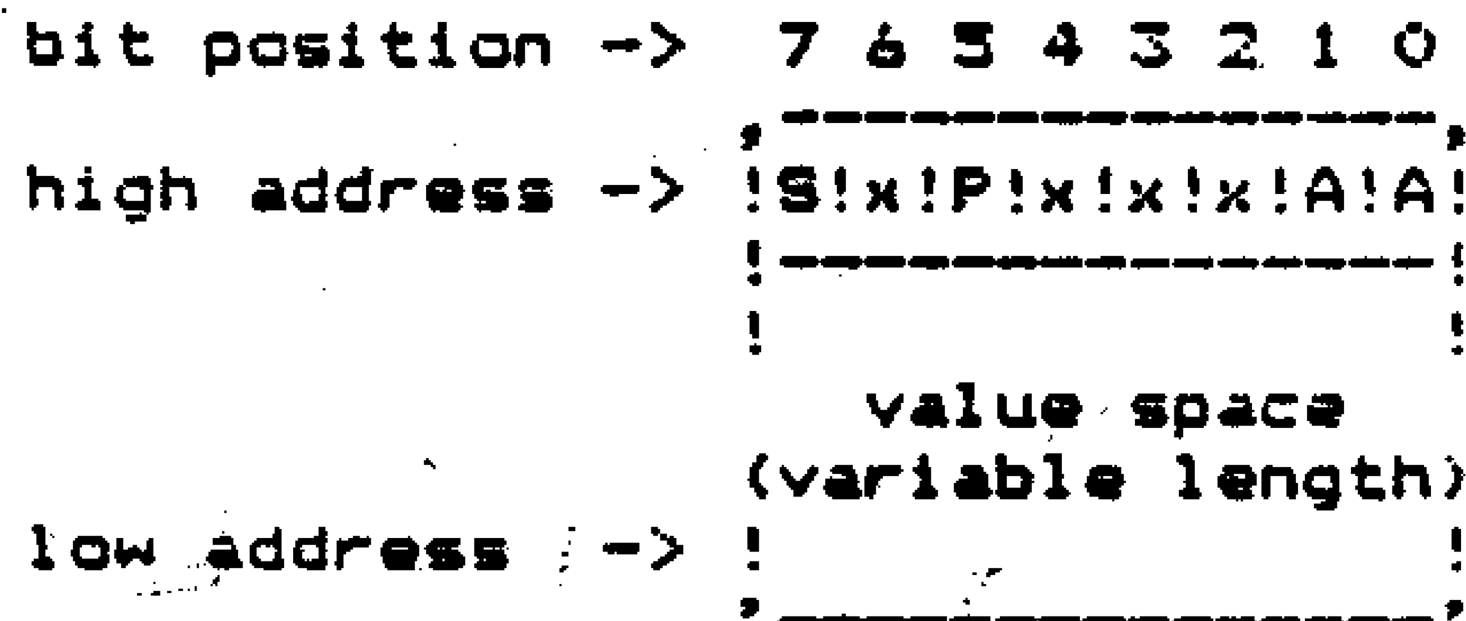
Checking for the End-of-Statement Token

The final step in parameter passing is to verify that the CALL statement is properly terminated. The last character should

be the end-of-statement token. If any other character is present, the subprogram can report an "Illegal syntax" error.

PASSING VARIABLES AS PARAMETERS

Parameter passing often involves accessing BASIC variables. Each variable in a program has a corresponding entry in the program symbol table. This entry consists of a flag byte and a value space. The flag byte indicates the type (string or numeric), the scope (local or shared), and the complexity (simple or array) of the variable. The value space may contain either the actual value of the variable or information about the location of the value, as indicated by the contents of the flag byte. An entry in the symbol table has the following general form.



Byte 0 -> Flags

```
    Bit 7 = 0 -> numeric entry
          = 1 -> string entry

    Bit 5 = 0 -> local variable. The value of this variable
                 is in this symbol table.
          = 1 -> shared variable. The value is in a different
                   symbol table.  The following value space
                   contains a pointer to the value space in the
                   other symbol table.

    Bits 1-0 = 00 -> simple variable
             = 01 -> one-dimensional array
             = 10 -> two-dimensional array
             = 11 -> three-dimensional array

    Bits 2,3,4, and 6 are reserved
```

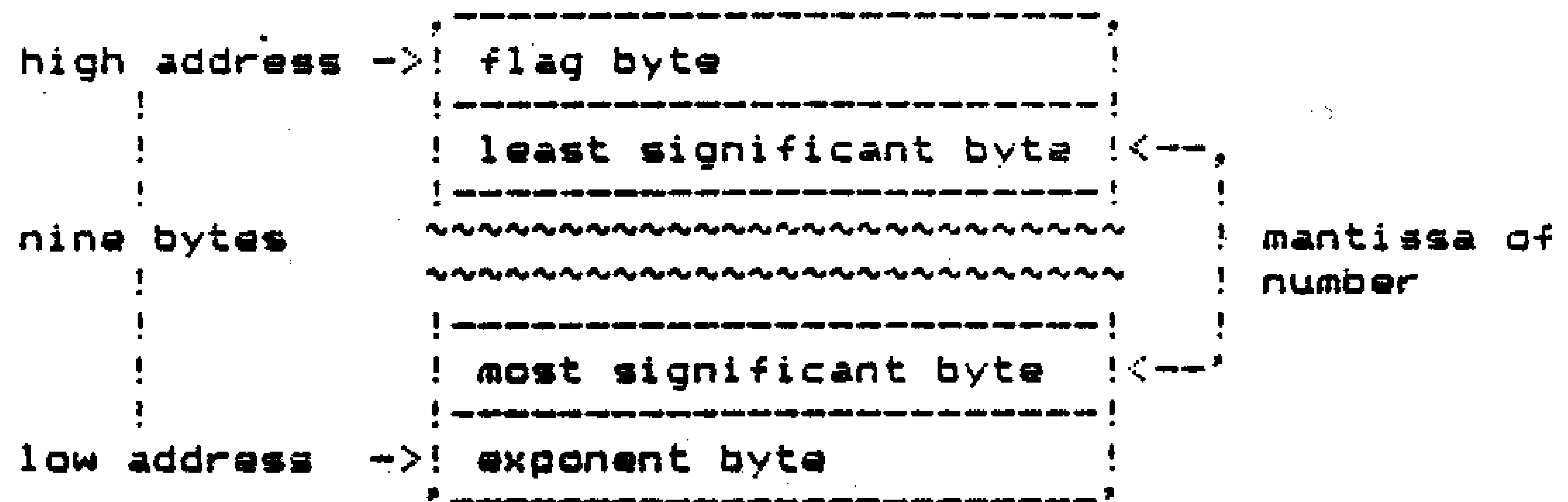
Bytes 1+ -> Value space

The contents and the length of this space are dependent upon the type of symbol table entry.

```
    Numeric -> 8 byte floating point value with the least
               significant byte at the highest address

    String  -> 2 byte pointer to the length byte at the high
               end of the string value

    Arrays  -> 2 byte number of elements for each dimension
               (number of elements = maximum subscript + 1)
               Then either

               * 8 bytes for each floating point number or

               * 2 bytes for each string pointer

    Shared  -> 2 byte pointer to the value space (not the
               flag byte) of the variable in the original
               symbol table
```

SIMPLE NUMERIC VARIABLES

Each simple (non-array) numeric variable has a corresponding floating point format entry in program symbol table as shown in the following figure.



The only exception to this form occurs when the numeric variable is shared between a "calling" BASIC program or subprogram and the "called" BASIC subprogram. When the variable is shared (as indicated by a one in bit 5 of the flag byte), the value space contains a pointer to the value rather than the value itself. The pointer is located in the highest addressed two bytes of the value space with the least significant byte at the highest address. The other six bytes of the value space are ignored.

The system provides two routines for the purpose of handling numeric variables during subprogram parameter passing. The first (GETNUM) is used when only the value of the argument is required by the subprogram. The second (GETADR) is used when the subprogram must have access to a variable in order to alter its value.

GETNUM - Get a Numeric Value

The routine called GETNUM moves the value of the next argument from the argument list into the floating point accumulator.

The argument can be a numeric constant, variable, array element, or expression.

Errors which occur are reported directly to the BASIC error handling routine. For example, if the value is a string rather than a number, GETNUM pushes the string entry on the floating point stack, stores the "String-number mismatch" error code (>03) in the A register, and executes a TRAP 8 instruction.

When no errors occur, GETNUM returns the floating point number in the floating point accumulator (registers >75 through >7C). The character following the argument is in the current character register and the BASIC program pointer is positioned properly.

GETADR - Get the Address of an Argument

This routine returns a symbol information entry which describes the next argument in the list. The argument can be a simple numeric variable or a numeric array element. All errors are reported directly to the BASIC error handler. GETADR returns two copies of the symbol entry, one in the floating point accumulator and one on top of the floating point stack.

Numeric Symbol Information Entry

Register	Information
>75	Not used
>76	>00 -> Numeric ID byte indicates that the entry contains numeric information
>77,>78	Value pointer to the least significant byte of the floating point number
>79,>7A	Pointer to the flag byte of the variable in the

symbol table entry

>7B,>7C Not used

The subprogram must check the ID byte (register >76) to ensure that the entry is the correct type (GETADR, as described below, may also be used with string variables). The value pointer [>77,>78] can be used to access the current value of the symbol.

Like GETNUM, GETADR leaves the character following the argument in the current character register and the BASIC program pointer properly updated.

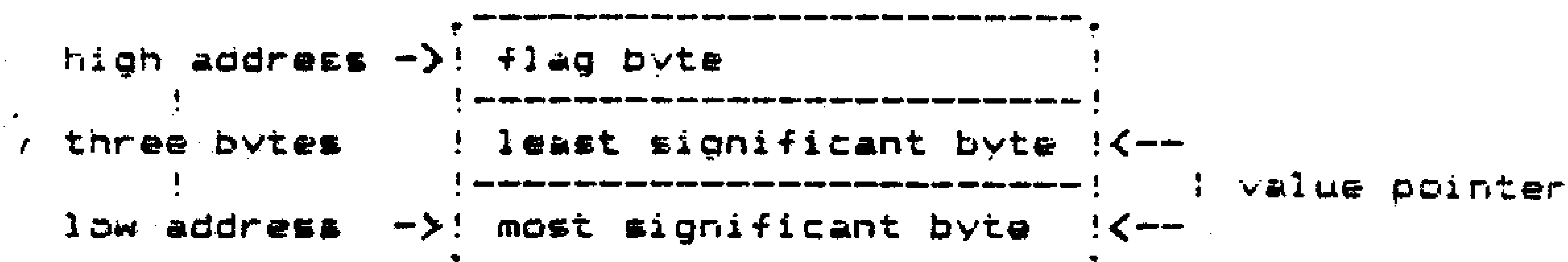
ASSIGN--Changing the Value of a Numeric Variable

The GETADR routine pushes the information entry on the floating point stack so that it can be used later to assign a new value to the variable.

Assignment of a new value to the variable uses the system ROM routine called ASSIGN. The new floating point value for a variable is placed in the floating point accumulator (registers >75 through >7C) while the symbol information entry is on the floating point stack. The ASSIGN routine verifies that the value and the symbol entry are the same type (both numeric); then it assigns the new value to the variable by copying it into the value space.

SIMPLE STRING VARIABLES

Each simple (non-array) string variable has a corresponding entry in the program symbol table as shown in the following figure.



If the string variable is local (bit 5 of flags = 0), the value pointer contains the address of the length byte of the string value assigned to the variable. If the string variable is shared (bit 5 of flags = 1), the value pointer points to the value space of the "original" variable in a separate symbol table. This "original" value space contains the pointer to the string value.

The string value is stored in reverse order with the length byte at the highest address. It resides either within the program image or within the dynamic memory area. Thus, string values must not be directly modified. If the string is in the program image, the program image can be destroyed. If the string is in the dynamic memory area, an unaltered length byte is necessary for correct operation of the dynamic memory management subsystem. Any alteration of this byte will cause that system to subsequently fail and destroy the contents of the dynamic area.

The system provides two routines for handling string arguments during parameter passing. The first (GETSTR) is used when the value of the argument is needed. The second (GETADR) is used when access to the argument is desired.

GETSTR - Get a String Argument

This routine gets the value of the next argument. The argument can be a quoted string constant, a string variable, an array reference, or an expression. Like GETNUM, GETSTR reports the occurrence of any errors directly to the BASIC error handling routine.

If no errors occur, GETSTR returns the following string-value information entry in the floating point accumulator.

String Value Information Entry

Register	Information
>75	Bit 7 = 0 -> The string value is temporary. = 1 -> The string value is assigned to a variable. Bit 6 = 0 -> The string value is in the program image. = 1 -> The string value is in the dynamic memory. Bits 5 through 0 are not used.
>76	>AA -> String ID byte indicates that the entry contains string information
>77,>78	Value pointer to the length byte of the string value

GETSTR also pushes the string information entry on the floating point stack for housekeeping purposes. When the subprogram is finished using the string value, it must pass the string information entry to one of the string clean up routines (STGCLR, STGCL2). If the string is temporary and resides in the dynamic memory area, the string clean up routine takes the necessary action to delete the string value from memory.

When GETSTR returns control to the subprogram, the current character register contains the character after the argument, and the BASIC program pointer has been advanced.

GETADR - Get the Address of an Argument

As with numeric variables, GETADR builds a symbol information entry describing the next argument. All errors are reported directly to the BASIC error handler.

The GETADR routine returns two copies of the symbol entry, one in the floating point accumulator and one on top of the floating point stack.

String Symbol Information Entry

Register	Information
>75	Bit 7 = 1 since the string is assigned to a variable Bit 6 = 0 -> The string value is in the program image. = 1 -> The string value is in the dynamic memory. Bits 5 through 0 are not used.
>76	>AA -> String ID byte indicates that the entry contains string information
>77,>78	Value pointer to the length byte of the string value
>79,>7A	Pointer to the flag byte of the symbol table entry
>7B,>7C	Pointer to the value pointer within the value space of the symbol table entry

The subprogram must check the ID byte (register >76) to ensure that the entry is for a string. The value pointer [>77,>78] can be used to access the current value of the string.

Like GETSTR, GETADR leaves the character following the argument in the current character register and the program pointer updated accordingly.

ASSIGN--Changing the Value of a String Variable

The GETADR routine pushes the information entry on the floating point stack so that it can be used later to assign a new value to the the string variable. The value assignment routine (ASSIGN) expects to be passed a string value information entry in the floating point accumulator (registers >75 through >7C) and a string symbol information entry on the floating point stack.

The ASSIGN routine verifies that the value and the symbol entry are both string type, releases the current value of the string variable, and assigns the new value to the variable.

String Creation and Deletion

Since string values can be up to 255 characters long, they are not handled in the register file. When BASIC assigns a string constant to a variable, the string value remains in the program image. When a string expression is evaluated or a string value is acquired from some source other than the program image (for example, from user input or from a file), the string value is placed in the dynamic memory area.

The dynamic memory management subsystem includes a routine (STGNEW) for the purpose of allocating space for a string value. The STGNEW routine is passed the desired string length; it returns a pointer to the length byte of the allocated string location. The string length is already stored in this location and must not be changed: it is used later by the deallocation

routine. The allocated space for the string characters precedes the length byte in memory. When STGNEW returns control to the calling subprogram, the the string characters must be copied into the allocated space.

The dynamic memory management subsystem also includes a routine (STGDSP) for deallocating space no longer required for a string value. The STGDSP routine is passed the pointer to the length byte of the allocated string space (the pointer originally obtained from STGNEW). This length byte must indicate the exact length requested when STGNEW was called. When control returns to the subprogram from STGDSP, the space previously used by the string value will be in the linked list of free memory blocks.

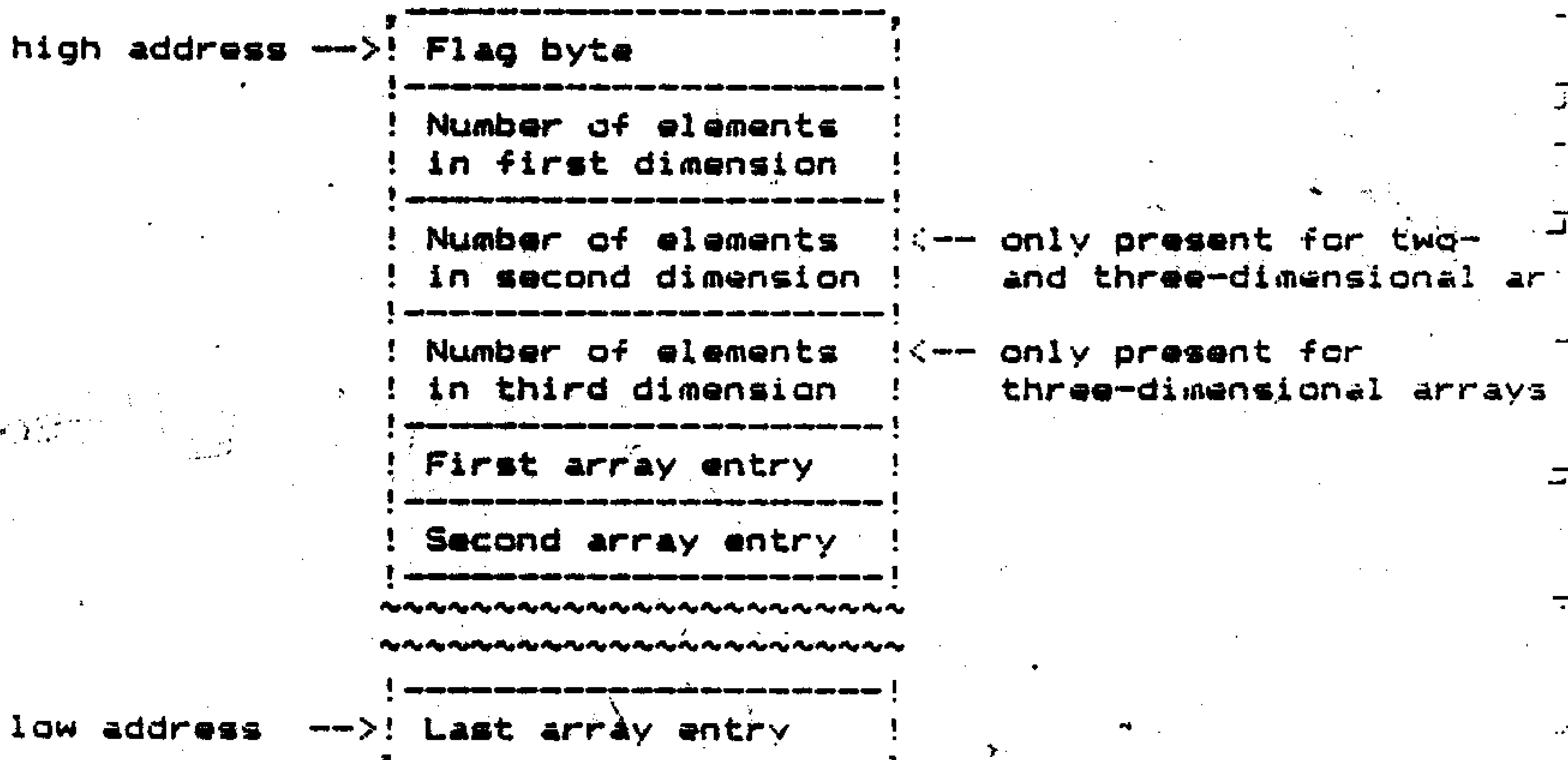
The STGDSP routine does not verify that the value pointer points into the dynamic memory area. If STGDSP is given a pointer into the program image, the program image will be destroyed. Two other routines in system ROM, however, verify that the string value to be released is in the dynamic memory area.

The two routines, STGCLR and STGCL2, differ from STGDSP in the information provided to them and in their checking the location of a string variable. Both routines are provided a string value information entry; for STGCLR, the entry is provided on the top of the floating point stack, and for STGCLR it is provided in the floating point second argument (registers >6B through >72). The STGCLR routine pops the entry from the floating point stack into the second argument registers and then enters STGCL2. Both routines verify that the value is temporary

and that it resides within the dynamic memory area. When these two conditions are met, the string space is returned to the linked list of free memory. If either of the conditions is not met, the routines return control to the calling subprogram without altering dynamic memory.

ARRAY VARIABLES

Each array has a corresponding symbol table entry. The length of the symbol table entry depends upon the number of dimensions assigned to a variable (1, 2, or 3) and the type of variable (string or numeric). The ordering within an array entry is shown in the following figure.



The number of elements in each dimension is stored with the least significant byte of the integer in the higher addressed byte and the most significant byte of the integer in the lower addressed byte.

For string arrays, each entry is a two byte value pointer that points to the length byte of the string value for the corresponding array element in dynamic memory. For numeric arrays each entry is the eight byte floating point value of the corresponding array element.

Elements in arrays are ordered so that the first dimension has the "least significance" in the sequence of entries in the table. For a one-dimensional array, elements assigned to each subscript appear in memory in ascending order of the subscripts. For a two-dimensional array, the elements in ascending order of subscripts in the first dimension appear just as they would in a one-dimensional array; following these elements, however, are all of the first-dimension elements for the next higher subscript in the second dimension. The following table shows the ordering of elements for one-, two-, and three-dimensional arrays that have three elements in each dimension.

Ordering of Array Elements

Dimension(s)	1	2	3
	(0)	(0,0)	(0,0,0)
	(1)	(1,0)	(1,0,0)
	(2)	(2,0)	(2,0,0)
		(0,1)	(0,1,0)
		(1,1)	(1,1,0)
		(2,1)	(2,1,0)
		(0,2)	(0,2,0)
		(1,2)	(1,2,0)
		(2,2)	(2,2,0)
			(0,0,1)
			(1,0,1)
			(2,0,1)
			(0,1,1)
			(1,1,1)
			(2,1,1)
			(0,2,1)
			~
			~
			~
			(1,2,2)
			(2,2,2)

When an array is shared (bit 5 of flags = 1), the value space contains a pointer to the value space of the "original" array variable. The pointer is stored in the highest addressed two bytes with the least significant byte at the highest address.

Passing Elements of an Array

Individual array elements are passed like simple variables. A CALL statement of the following form is used to pass the value of the variable subscripted X,Y,Z.

```
CALL SUBPROG(VARIABLE_NAME(X,Y,Z))
```

As with simple variables, the subprogram should then check syntax and pass the value of the element through the GETNUM, GETSTR, or GETADR routine.

Accessing an Entire Array

If an entire array, rather than a single element in an array, is to be passed as an argument (called a "formal" array argument), a system ROM routine called SYM must be used.

The SYM routine locates the symbol table entry for a formal array argument in the argument list. A formal array argument is the array name followed by a set of parentheses and commas to indicate the number of array dimensions. For example, a one dimensional array named PROMPT$ appears in the argument list as PROMPT$(), a two dimensional numeric array named POINTS appears as POINTS(,), and a three dimensional array named COORDINATES appears as COORDINATES(,,).

The SYM routine returns a pointer [>79,>7A] to the flag byte of the symbol table entry corresponding to the name of the variable. This pointer can be used to check the flag byte for the desired variable characteristics (type, number of dimensions, and so on) and then to access the desired value(s).

The SYM routine does not advance the BASIC program pointer to check the syntax of the CALL statement argument. The machine-language subprogram must verify that the set of commas in the argument list corresponds to the number of dimensions indicated in the flag byte of the array. The subprogram program must advance the program pointer, character by character, using the GETCHR routine and check for the appropriate set of tokens, that

is, () for a one-dimensional array, (,) for a two-dimensional array, or (,,) for a three-dimensional array.

Locating Elements in an Array

Once the location of the first element in an array is known and the array is verified against the argument, the locations of individual elements in the array must be calculated. The following formulas can be used to calculate the addresses of individual array entries within the value space.

Definition of symbols

subscript1 ->	first subscript of one-, two-, or three-dimensional array element
subscript2 ->	second subscript of two- or three-dimensional array element
subscript3 ->	third subscript of three-dimensional array element
dimension1 ->	number of elements in the first dimension of a one-, two-, or three-dimensional array (stored in symbol table entry)
dimension2 ->	number of elements in the second dimension of a two-, or three-dimensional array (stored in symbol table entry)
entry_size ->	two bytes for string arrays, eight bytes for numeric arrays

With these definitions, the offset from the first array entry to the desired array entry is calculated as follows.

```
One-dimensional array:
     entry_size * subscript1

Two-dimensional array:
     entry_size * ( subscript1 + ( subscript2 * dimension1 ) )

Three-dimensional array:
     entry_size * ( subscript1 + ( dimension1
                        * ( subscript2 + ( subscript3 * dimension2 ) ) ) )
```

Subtracting the offset from the address of the first array entry provides the address of the desired array entry.

Value Assignment for an Array Element

Assignment of a new value to an element of a numeric array can be accomplished in one of two ways.

One method is direct modification of the array's value space; that is, after an array entry is located, and a new value is copied into that location.

The other method uses ASSIGN. The new value of the element is moved into the floating point accumulator and the numeric symbol information entry is pushed onto the top of the floating point stack. When ASSIGN is called, it performs the necessary type checking and assigns the new value.

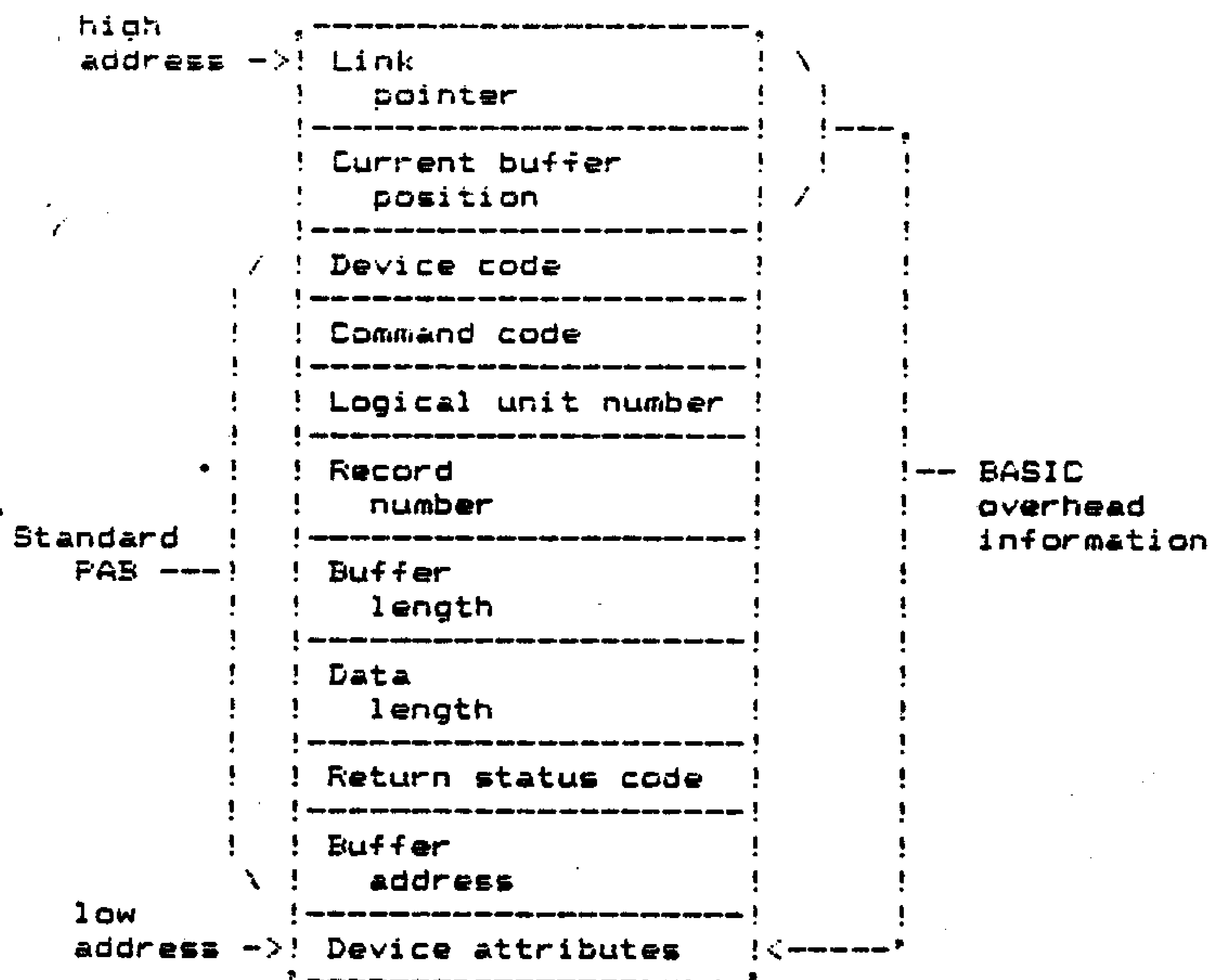
Assignment of a new value to an element of a string array is more complex. Direct modification of the value space does not eliminate the old value as it does with numeric entries. If the old value resides in the dynamic memory area, it must be released using the STGDSP routine before the value pointer in the array entry is destroyed. After the string value has been released, dynamic memory space for the new value to be moved into can be

allocated using STGNEW, the new value moved into the space, and pointer to the new value stored in the array entry.

Alternately, the ASSIGN routine can be used to assign a new value to a string array element. A string symbol information entry for the array element must be placed on top of the floating point stack and a string value information entry must be placed in the floating point accumulator. The ASSIGN routine performs the necessary type checking, releases the current value of the string array element, and assigns the new value.

PERIPHERAL I/O

The BASIC operating environment uses the I/O subsystem, described in the previous chapter, to perform all communications with peripheral devices. BASIC maintains a peripheral access block (PAB) for each open device or file. These active PABs are kept in a linked list in the dynamic memory area. Associated I/O buffers are also located in the dynamic memory area. BASIC adds five bytes of overhead to each PAB to maintain the linked list and the context of each open device or file. A PAB maintained by BASIC is organized as shown in the following figure.



The link pointer is stored with the least significant byte of the value at the highest addressed byte. It points to the least significant byte of the link pointer in the BASIC PAB which is next in the linked list. The link pointer is zero for the last PAB in the list. BASIC maintains a pointer [>8EE,>8EF] to the first PAB in the list. When the list is empty, this pointer is zero. Although BASIC does use PABs in the register file, it never links a register file PAB into the list. Therefore, only the most significant bytes of the first PAB pointer [>8EE,>8EF] and the individual link pointers must be checked for zero to determine the last PAB.

The current buffer position is a pointer into the I/O buffer

and is maintained to permit BASIC to allow "pending" I/O. In other words, BASIC allows more than one program statement to access a single record of data. Between successive I/O statements, BASIC maintains the current position within the I/O buffer so that the next read or write can correctly access the data record.

The device attributes byte is a copy of the device attributes byte that is sent to the peripheral device during the open operation. BASIC uses the flags in this byte to communicate properly with the device or file.

BASIC uses the the statement level temporary area (registers >3A through >4A) to build and manipulate PABs. By using these registers, BASIC can take advantage of faster and more powerful processor instructions in manipulating the PAB than it could by using the system RAM.

In using a PAB, BASIC creates it, manipulates it, and then closes it.

A PAB is created in the statement level temporary area from the information included in an OPEN statement. After the device is successfully opened, space is allocated through the dynamic memory manager for the PAB and the associated I/O buffer. Finally, the PAB is copied into the dynamic area and linked into the list.

Each time a peripheral I/O statement (e.g. PRINT # or INPUT #) specifies the logical unit number corresponding to the device or file, the PAB is located and copied back into the statement level temporary area. Any necessary modifications are made, and

the I/O call is performed. Upon successful completion of the I/O call, the PAB is copied back to its location in the dynamic area.

When the CLOSE statement is executed, the PAB is copied into the statement level temporary area. The close I/O call is performed. After successful completion of the close operation, the dynamic memory manager is called to release the memory space occupied by the PAB and the associated I/O buffer.

OPENING DEVICES OR FILES

System ROM includes a subroutine (OPEN) that can be useful when a subprogram opens a device or file. The subprogram must provide the device attributes byte, the requested I/O buffer length, the unique logical unit number, and a string entry. The ASCII string must contain the device or file specification as it would appear in a BASIC OPEN statement, in the following form.

device.filename

where

device = the unique device code associated with the peripheral device

filename = the device options field that contains either a file name for a mass storage device or device options settings for other devices.

A call to the OPEN subroutine then performs the following tasks.

1. The remaining parts of the PAB in the statement level temporary area are released.
2. The OPEN I/O buffer is allocated.
3. Required information, including translation of lower case alphabetic characters to upper case within the device options field, is stored in the buffer.

4. The device or file specification string is released
5. An OPEN I/O call is performed.
6. The returned information is moved into the PAB.
7. The OPEN I/O buffer is released.
8. The "temporary PAB" flag is set.
9. Control returns to the subprogram.

If an I/O error occurs during the attempted open operation, the OPEN routine TRAPs to the error handler after releasing the OPEN I/O buffer but before setting the temporary PAB flag.

If the device or file is to remain open in the BASIC operating environment, the subprogram must perform the following steps.

1. An I/O buffer of the accepted size must be allocated.
2. The buffer pointer in the PAB must be initialized.
3. Dynamic area space for the PAB must be allocated.
4. The PAB must be copied into its allocated space.
5. The PAB must be entered into the linked list.
6. The "temporary PAB" flag must be reset.

The temporary PAB flag signals that the statement level temporary area contains a PAB for an open device or file and that the PAB has not been linked into the PAB list yet. If an error occurs before the PAB is linked into the list, the error handler can therefore use the PAB in the statement level temporary area to close the device or file.

CLOSING DEVICES OR FILES

System ROM includes two subroutines (CLSTMP, CLSALL) that can be useful when closing devices or files. The CLSTMP routine checks the temporary PAB flag. If the flag is set, the routine assumes that the statement level temporary area contains a PAB for an open file or device that is to be closed. CLSTMP stores the close I/O command in the PAB, clears the data length, issues the I/O call, and returns to the calling program. If an I/O error occurs during the attempted close operation, CLSTMP sends an I/O warning to the BASIC error handler. After the warning is acknowledged, control returns to the CLSTMP routine, and then, to the calling program.

The CLSALL routine closes all open files and devices in the linked list of PABs. If a temporary PAB is present in the statement level temporary area, the assiciated file or device is also closed. First, CLSALL calls CLSTMP for any file or device with a temporary PAB. Next, CLSALL executes a loop that repeatedly performs the following tasks until all devices and files are closed.

1. The next PAB in the list is sought out.
2. If necessary, any pending output to a device or file is sent.
3. The device or file is closed.
4. Dynamic memory for the PAB and associated I/O buffer is released.
5. Steps 1 through 4 are performed until the last PAB is processed

Any I/O errors that occur during this process are reported to the BASIC error handler as warnings rather than errors. After the user acknowledges a warning, control returns to the CLSALL routine, and it continues processing with the next PAB.

ERROR PROCESSING

In the BASIC operating environment, errors and warnings are processed by placing the appropriate code value in the A register and executing a TRAP >08 instruction. This instruction transfers control to system RAM location >81C which contains a branch instruction to the BASIC error handler routine. The actions taken by the error handler depend upon the value in the A register and the current status of the BASIC ON WARNING and ON ERROR statements.

The value of the error code in the A register indicates one of three types of errors to the error handler. If the value is negative, the error is a warning. If the value is positive, the error is a fatal error.

The BASIC error handler routine processes warnings and fatal errors in accordance with control instructions provided by the program being run. The default status settings of the corresponding BASIC control statements are ON WARNING PRINT and ON ERROR STOP. The other warning options (ON WARNING NEXT, ON WARNING ERROR) allow a BASIC program to either ignore warnings or treat them as fatal errors. A second error option (ON ERROR line-number) allows a program to process errors (including warnings treated as errors) with routines written in BASIC language.

ON WARNING PRINT

When the error handler receives a warning code under this condition, it takes the following actions.

1. It saves 36 registers (>5C through >7F) from the function level temporary area in the line compression buffer (>925 through >948).
2. It outputs the error message to the display and turns on the error indicator.
3. It awaits user acknowledgement of the message.
4. It restore the 36 registers in the function level temporary area.
5. Control return to the calling program.

If the user presses the BREAK key when the error handler is waiting for acknowledgement, the error handler calls the breakpoint routine instead of returning control to the calling program. If the user presses the OFF key, the error handler transfers control to the power-down (POWDWN) routine.

ON WARNING NEXT

When the error handler receives a warning code under this condition, it does nothing but return control the calling program. In effect, the warning is ignored.

ON WARNING ERROR

When the error handler receives a warning code under this condition, it resets the most significant bit of the warning code to change it from a warning code into an error code. The result is that the error handler processes the warning as an error as described below.

ON ERROR STOP

When the error handler receives an error code under this condition, it displays the appropriate error message and turns on the error indicator. It closes all open devices and files that are included in the linked list of PABs and, if necessary, any device and file associated with the temporary PAB that is in the statement level temporary area. If the error is "Memory full," the error handler reinitializes the dynamic memory area. Finally, the error handler transfers control to the command level of the system. If the user has acknowledged the error message by pressing the OFF key, the error handler transfers control to the power-down (POWDWN) routine.

ON ERROR Line-Number

When the error handler receives an error code under this condition, it prepares to transfer control to the designated error-handling BASIC subroutine. The preparation consists of placing on the floating point stack an entry that contains the current BASIC execution context. If the error handling subroutine terminates with a RETURN or a RETURN NEXT statement, the information in the stack entry is used to transfer control to the appropriate position in the BASIC program. After the error handler places the execution context on the floating point stack, it transfers control to the error handling subroutine.

REDIRECTION OF THE ERROR HANDLER

A machine language program or subprogram can redirect all error handling by storing a branch instruction in the error-

handler vector location in system RAM (>81C). If redirection is performed by a program running within the BASIC operating environment, the standard vector must be moved back into place before control is returned to the BASIC system.

BREAKPOINT PROCESSING

The system provides a subroutine (BRKPT) that is used to report the occurrence of a breakpoint when running in the BASIC operating environment. If a program detects the BREAK key down by calling one of the keyboard control routines, it should call the BRKPT subroutine. BRKPT checks the status of the ON BREAK statement from BASIC. ON BREAK STOP is the default. If the default is in effect, BRKPT calls the error handler to report the breakpoint message. Then, the current program execution context is placed in an entry on the floating point stack. When the user enters a CONTINUE command, the information in the breakpoint stack entry is popped to resume execution of the program.

The other options (ON BREAK NEXT and ON BREAK ERROR) can be used to ignore breakpoints or to treat them as errors. If ON BREAK NEXT is selected, the BRKPT routine returns control to the calling program with no other action, thus ignoring the breakpoint. If ON BREAK ERROR is selected, the BRKPT routine calls the error handler to process the breakpoint as an error rather than a simple message.

MISCELLANEOUS SYSTEM ROUTINES

The system provides four other potentially useful subroutines that can be called by programs developed through

assembly language. The following table lists the routines and the functions they perform.

Routine	Function
NEWPRO	Initializes the main program area by overwriting the current contents with an empty BASIC program and header.
TRSHDY	Initializes the dynamic area and associated pointers. The base addresses for the dynamic area and the floating point stack are assumed to be correct.
TYPE	Determines the type (alphabetic or numeric) of the character in register >5C.
VERSION	Returns a one-byte numeric value that indicates the resident version of BASIC.