PRODUCT 359
BASIC INTERPRETER
Design Specification

Personal Computer Division

June 30, 1980

Revision 3.0

Change Status

page	level	page	level	page	level
1	3.0	41	3.0	81	3.0
2	3.0	42	3.0	82	3.0
3	3.0	43	3.0	83	3.0
4	3.0	44	3.0	84	3.0
5	3.0	45	3.0	85	3.0
6	3.0	46	3.0	86	3.0
7	3.0	47	3.0	87	3.0
8	3.0	48	3.0	88	3.0
9	3.0	49	3.0	89	3.0
10	3.0	50	3.0	90	3.0
11	3.0	51	3.0	91	3.0
12	3.0	52	3.0	92	3.0
13	3.0	53	3.0	93	3.0
14	3.0	54	3.0	94	3.0
15	3.0	55	3.0	95	3.0
16	3.0	56	3.0	96	3.0
17	3.0	57	3.0	97	3.0
18	3.0	58	3.0	98	3.0
19	3.0	59	3.0	99	3.0
20	3.0	60	3.0	100	3.0
21	3.0	61	3.0	101	3.0
22	3.0	62	3.0	102	3.0
23	3.0	63	3.0	103	3.0
24	3.0	64	3.0	104	3.0
25	3.0	65	3.0	105	3.0
26	3.0	66	3.0	106	3.0
27	3.0	67	3.0	107	3.0
28	3.0	68	3.0	108	3.0
29	3.0	69	3.0	109	3.0
30	3.0	70	3.0	110	3.0
31	3.0	71	3.0	111	3.0
32	3.0	72	3.0	112	3.0
33	3.0	73	3.0		
34	3.0	74	3.0		
35	3.0	75	3.0		
36	3.0	76	3.0		
37	3.0	77	3.0		
38	3.0	78	3.0		
39	3.0	79	3.0		
40	3.0	80	3.0		

Table of Contents

1.0 Introduction

This document contains the design details and documentation of the Texas Instruments Product 359 BASIC language processor.

1.1 Purpose

The information provided in this document is intended to provide a comprehensive documentation of how the Product 359 BASIC interpreter functions and is intended to be a document which can be referenced by persons maintaining, modifying or using the interpreter on a very intimate level. Included are descriptions of the use of memory, flow of the BASIC interpreter and information deemed necessary to illuminate all facets of the interpreter. This document was used as the design tool for the implementation of the Product 359 BASIC interpreter.

1.2 Scope

The information contained herein is intended to be a complete view of the interpreter for the implementers of the BASIC as well as for persons maintaining the BASIC. It contains all information about the interpreter exept that information which is contained in the documents named in section 2.

2.0 Applicable Documents

TI Extended BASIC Language Specification (June 30, 1980)

Product 359 BASIC Subprogram Specification (June 30, 1980)

Product 359 BASIC Sprite Specification (June 30, 1980)

Product 359 BASIC Interpreter Expansion RAM Peripheral Support Software Specification (June 30, 1980)

Product 359 BASIC Language Implementation and Verification Specification (June 30, 1980)

Home Computer BASIC Language Specification (April 14, 1979)

Specification of a Texas Instruments Standard for the BASIC Language (June 9, 1978)

American National Standard for Minimal BASIC (ANSI X3.60-1978)

TMS 9900 Microprocessor Data Manual (December 1976)

Graphics Programming Language Programmers Guide (June 1, 1979)

Top Down Operator Precedence (Vaughan R. Pratt, ACM Symposium on Principles of Programming Languages, Boston, Mass., October 1973)

3.0 General Description

This BASIC interpreter is to be contained in a Solid State Software (TM) Module that can be plugged into any 99/4 Home Computer console. The interpreter is intended to be ANSI and TI Standard compatible and is intended to provide access to some of the unique features of the 99/4's hardware, as well as the expansion memory peripheral. The interpreter provides several enhancements above the ANSI nucleus and 99/4 BASIC to access the color graphic, sprite, and sound capabilities of the 99/4 console as well as access to the expansion memory and speech peripherals. This BASIC is considered to be an "enhanced" or "extended" BASIC for the 99/4 console owner who needs a more powerful BASIC language interpreter.

3.1 Use of System Resources

This section describes how the interpreter utilizes the memory resources of the 99/4 console, including the RAM and ROM contained within the system, the GROM and ROM within the Product 359 Software Module and the expansion memory peripheral. The interpreter, itself, resides in 12K of ROM and 18K of GROM within the Software Module. It also utilizes the floating-point routines and some of the conversion routines contained within the 99/4 console software. As much of the 99/4 BASIC code as possible was translated into 9900 Assembly Language and put into high-speed ROM to increase the performance of the interpreter.

This BASIC utilizes the VDP RAM for program storage, symbol table storage, Peripheral Access Blocks, string space, crunch buffer, sprite-access blocks, screen and the floating-point stack. When the expansion memory peripheral is attached to the console, it is utilized for program storage and numeric variable storage with the other structures remaining in the VDP RAM.

The CPU RAM is used to maintain all of the necessary pointers and temproary variables involved with editing, prescanning and executing a BASIC program.

3.1.1 ROM Usage

ROM utilization by the Product 359 BASIC interpreter consists of two separate sections. First, and foremost, is the ROM contained in the Software Module. The ROM contained, therein, contains the bulk of the interpreter, including the parser, run-time support routines, as well as some of the speed-critical code of the editor and the static scanner. The ROM code contained in the Software Module is distributed between twenty-five separate assemblies. These assemblies are entitled XML359, REF359, BASSUP, STRING, PARSE, NUD, FORNEXT, SCROLL,

GREAD, GWRITE, DELREP, MVDN, MVUP, VGWITE, GVWITE, GETPUT, CNS, TRINSIC, SPEED, CIF, SCAN, CRUNCH, CPT, GETNB and SUBPROG. CRUNCH contains the input-line crunch routine with CPT being the Character Property Table used to interpret the incoming characters and GETNB being the GET Non-Blank routine used by CRUNCH. PARSE, NUD and FORNEXT contain the bulk of the parser, with NUD and FORNEXT existing solely because PARSE is a large assembly. BASSUP includes some of the important run-time support routines, like the symbol table searching routine, the array subscripting routine and the variable assignment routine. STRING contains the bulk of the system string handling facilities. TRINSIC includes the trignometric functions, square-root function, logarithm and exponential functions, and the involution routine. SUBPROG contains the run-time support for calling and executing BASIC subprograms correctly. It also contains the static-scan-time code for resolving all subprogram references. SCAN constains the 9900 support code for the static scanner to improve it's speed. CNS contains the Convert Number to String routine. It contains support for both free-format conversion of numerics and fixed-format conversion of numerics for use by the print/display-using-statements. CIF contains the Convert Integer to Floating routine which converts a 2-byte integer into its equivalent 8-byte radix 100 value. XML359 contains equates for routines in the 99/4 console code, equates for memory locations used in the on-board CPU RAM and the two XML tables used by the interpreter. The remainder of the assemblies contain special code to access the expansion memory from GPL and assembly language.

Physically, the ROM within the Software Module resides in the 8K address space from address >6000 to >7FFF which is predecoded at the GROM-port connector and is reserved for "plug-in" software. In order to get 12K bytes in the 8K address space special hardware is utilized by the Product 359 BASIC.

The software has been divided into 3, 4K blocks for the purpose of fitting it into the Software Module. Block 0 resides at address >6000 and is continuously enabled. Blocks 1 and 2 both reside at address >7000 with only one of the two blocks enabled at a particular time.

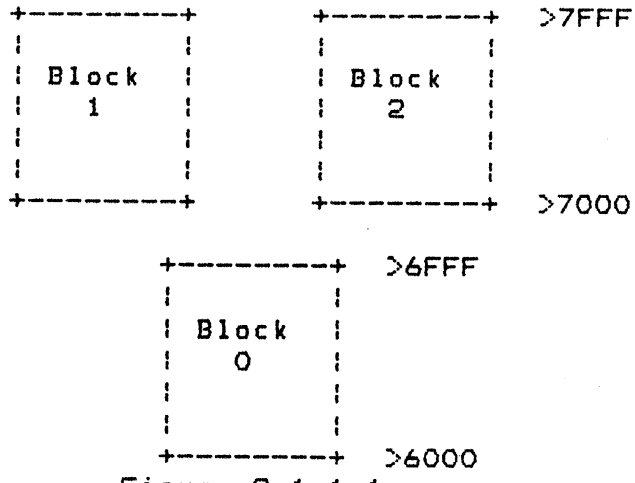


Figure 3.1.1.1

Upon powerup of BASIC block 1 is the block enabled by the software by writing to location >6000. In order for the software to enable and use block 2 it must write to location >6002.

Write to >6002 - block 2 is enabled

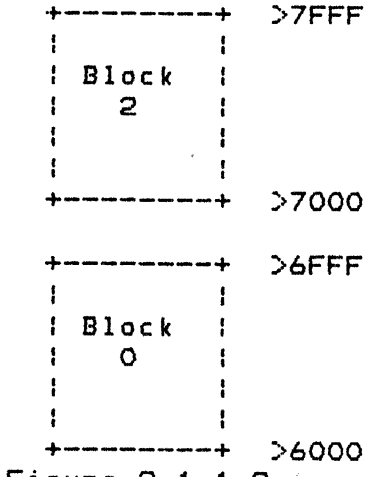


Figure 3.1.1.2

Once block 2 has been enabled, block 1 cannot be accessed. In order for the software to enable and use block 1 it must write to location >6000. When block 1 is enabled, block 2 cannot be accessed.

Write to >6000 - block 1 is enabled
(and on powerup)

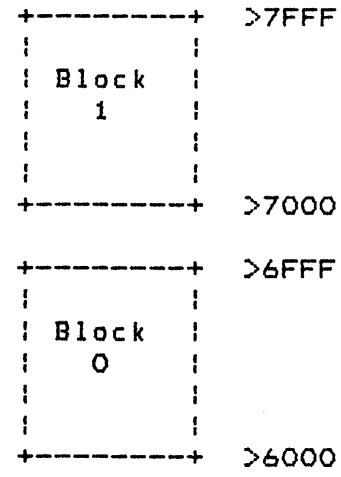


Figure 3.1.1.3

As can be seen, software in block 1 cannot access software in block 2 and vice versa. For this reason, the Convert Number to String (CNS) routine and the scientific intrinsic functions and CRUNCH have been placed into block 2 and are only invoked by code in block 0. CNS, CRUNCH and the intrinsic functions were chosen because they are called in relatively few places, they invoke no additional routines and the routines, themselves, are relatively large, allowing them to fill block 2.

In order to put together all of the ROM code for the Product 359 BASIC, each of the separate modules of the system must be included in two separate link-edits. The modules contained in the fist link-edit include all of the modules separated as shown below:

Block 0 at >6000
- XML359 - XML tables and equates
- REF359 - linkage to console
- CPT - Character Property Table
- BASSUP - the BASIC support package
- PARSE - the BASIC parser
- GETPUT - GET and PUT bytes from/to VDP and ERAM
- NUD - assorted NUD and LED handlers
- SPEED - speed-up routines for GPL subprograms
- MVUP - move forwards in memory
- GETNB - get non-blank for crunch

Block 1 at >7000
- FORNEXT - the FOR and NEXT statement handlers
- STRING - extended string package
- CIF - convert integer to floating
- SUBPROG - CALL, SUBEXIT, and SUBEND handlers
- SCROLL - screen scroll, other small routines
- SCAN - static scan dispatcher code

```
        GREAD   - read ERAM into CPU
        GWRITE  - write CPU to ERAM
        DELREP  - program line-delete
        MVDN    - move backwards in memory
        VGWITE  - move from VDP to ERAM
        GVWITE  - move from ERAM to VDP
Block 2 at >7000
        CNS     - the convert number to string routine
        TRINSIC - the trig and asst intrinsic package
        CRUNCH  - input-line crunch
```

The second link-edit contains all of the modules listed in the first link-edit except that the modules in block 2 are left out. When the code is loaded into a simulator or into an EPROM burner the code in the second link edit is used to get the code in block 1 only. When the code contained in the first link-edit is loaded the code in block 1 is over-written by the code in block 2 and therefore the fist link-edit supplies the code in blocks 0 and 2.

The ROM code within the 99/4 console is utilized in two ways. First, the GPL interpreter is used to interpret the portions of Product 359 BASIC which are written in GPL. Second, the floating point routines of the console, and some of the conversion routines of the console are used to minimize the amount of code which appears in the Software Module and to optimize the speed of the BASIC.

3.1.2 GROM Usage

The GROM portion of the interpreter contains a substantial portion of the BASIC interpreter. The line-input, program editing and static-scanning routines are contained in the GROM. Some of the statement, NUD and LED handlers are also contained in GROM. All input and output routines and all of the GPL subprograms such as SOUND, COLOR and SPRITE are located in GROM. The error handling routine as well as the error messages' text are contained in the GROM code.

The GROM portion of the interpreter is contained in five separate assembly modules. These modules are entitled, EDIT, PSCAN, EXEC, FLMGR and SUB and each of the assemblies handles the portions of BASIC described by these nmemonics. EDIT handles mainly the editing, commands and top-level portions of BASIC. PSCAN contains the static scan routine as well as assorted subroutines such as the line-input routine and the error handling routine. EXEC contains all of the GPL portion of execution except the screen / file management portions which are contained in the FLMGR assembly. SUB contains all of the sprite-access subprograms as well as the speech-access

subprograms.

In order to include all of the separate assemblies of the BASIC together the GPL object code linker is used. The linker is, reduced to simple terms, a file concatenator. There is no link-editing capability so that linkage from one assembly to another is fairly complicated. This BASIC gets from one assembly to another by using branch-tables which are set up at the beginning of each assembly, 'ORG'ing them to permanent locations. In order to tranfer control between routines in different assemblies, the address of the branch-table entry for the destination routine is used as an equate in the assembly of the source routine. An example would be:

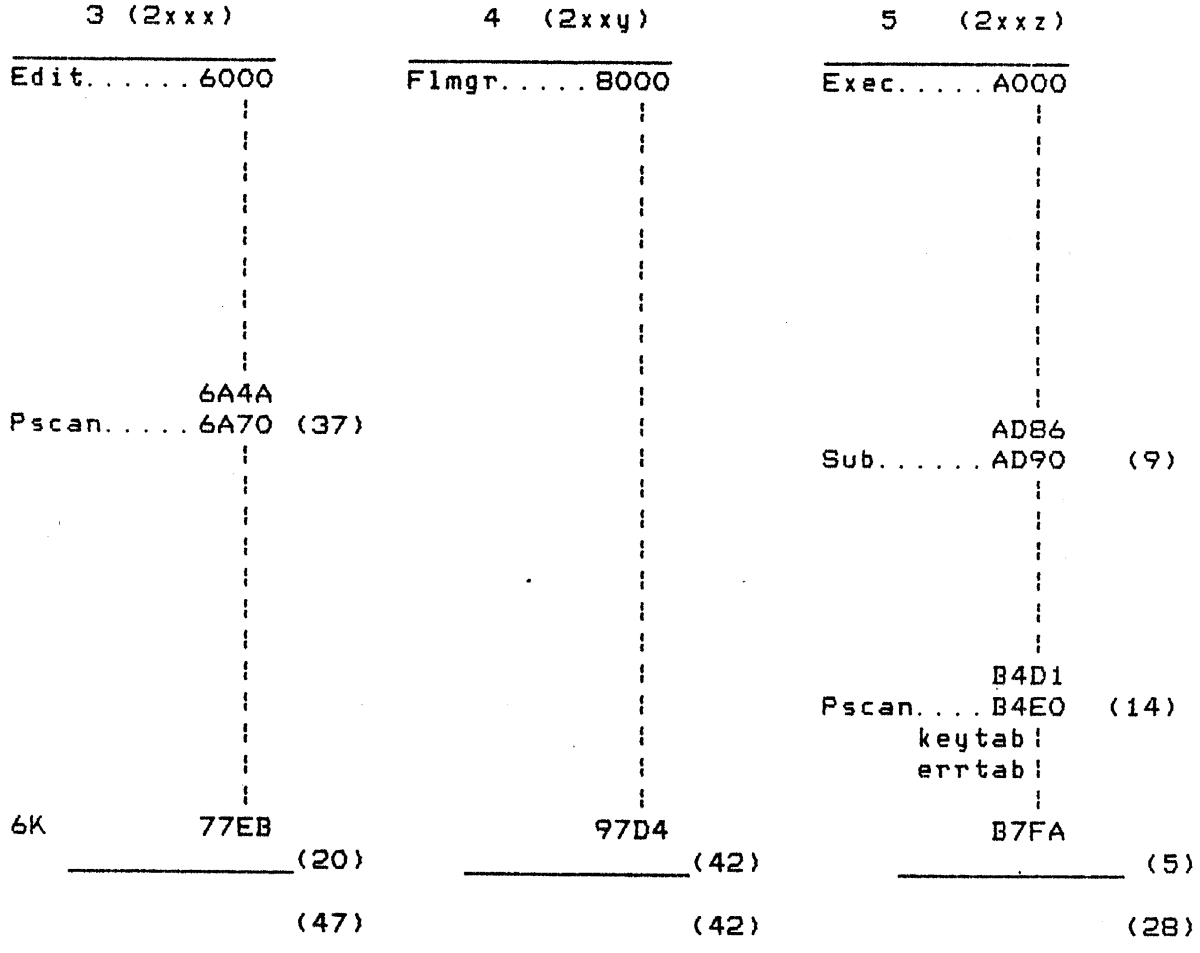
```
*       ASSEMBLY 1
*       BRANCH TABLE
        GROM 4
        ORG >40
        BR    READLN            >8040 - Link to line input routine
        BR    PRESCN            >8042 - Link to prescan routine

*       ASSEMBLY 2
READLN  EQU   >8040             Address of link to scan
WRITE   EQU   >8042             Address of link to write
        CALL  READLN            Read an input line
        B     PRESCN            Prescan the program
```

In this example, assembly 2 has the equates for the routines contained in assembly 1 and uses them in the CALL and Branch statements. It should be noted that this causes one extra instruction (the table entry Branch instruction) to be executed for a call from one module to another.

In order to keep track of how much of the GROM is being used and which parts are used by which sections, a GROM MAP is kept reflecting the organization of the console GROMs. The following is the GROM MAP for the latest revision of the Product 359 GROM code.

PRODUCT 359 GROM MEMORY MAP

06/30/80 10:45 (SRH)



Available per GROM:

GROM 3	47
GROM 4	42
GROM 5	28
Total	117

3.1.3 CPU RAM Usage

The first 140 bytes of the 256 bytes of on-board RAM are used exclusively by the BASIC interpreter. The remaining bytes are used by the Graphics Language interpreter and various interrupt routines and peripheral devices. This section itemizes BASIC's usage of the first 140 bytes and generalizes the GPL interpreter's usage of the remaining bytes. Note that the GPL interpreter's workspace (>E0 to >FF) is also used by the

Assembly Language portions of BASIC, so care must be taken to not destroy anything that the GPL interpreter needs preserved. This includes workspace registers 13, 14 and 15.

Address	Name	Use
00-17		BASIC statement temporaries
18-19	STRSP	start of string space pointer (high address)
1A-1B	STREND	end of string space pointer (low address)
1C-1D	SREF	temporary string pointer
1E-1F	SMTSRT	beginning of current statement being executed
20-21	VARW	screen display location start
22-23	ERRCOD	error vector for communication between GPL and assembly language portions of BASIC
24-25	STVSPT	base of value stack / top of character table-8
26-27	RTNG	return address for communication between GPL and assembly language portions of BASIC
28-29	NUDTAB	address of NUD table for NUDs handled by GPL
2A-2B	VARA	screen display location end
2C-2D	PGMPTR	program text pointer into a BASIC line
2E-2F	EXTRAM	line table pointer to line being executed
30-31	STLN	start of line table pointer(low address)
32-33	ENLN	end of line table pointer(high address)
34-35	DATA	pointer to current DATA element for READ stmts
36-37	LNBUF	pointer to line table for current DATA statement
38-39	INTRIN	address of intrinsic function's constants
3A-3B	SUBTAB	pointer to first subprogram table entry in chain
3C-3D	IOSTRT	pointer to first PAB in chain
3E-3F	SYMTAB	pointer to first symbol table entry in chain
40-41	FREPTR	pointer to free space to be used by symbol table
42	CHAT	character being processed by the interpreter
43	BASE	current option base of BASIC arrays
44	PRGFLG	program/statement execution mode
45	FLAG	general flag byte
46-47	BUFLEV	current destruction level of the crunch buffer
48-49	LSUBP	address of last subprogram block on the stack
4A-6D	FAC/ARG - floating-point accumulators	
6E-6F	VSPTR - Value-stack pointer	
70-71	highest address of VDP RAM	
72-7F	GPL status block	
72	DATA-stack pointer	
73	subroutine-stack pointer	
74	keyboard number for SCAN	
75	input character and math pack temporary	
76-77	joystick X and Y positions, math pack temporaries	
78	8-bit random number	
79	timer	
7A	sprite motion	
7B	VDP status register	
7C	GPL status register	
7D	character buffer for XPT and YPT	
7E	horizontal (X) screen pointer	
7F	vertical (Y) screen pointer	
80-83	unassigned at this time	

84	RAMTOP - highest address available in ERAM
86	RAMFRE - address of free space in ERAM
88	unassigned at this time
89	RAMFLG - ERAM/VDP flag
8A-BF	subroutine and data stack-space
C0-D9	GPL interpreter work areas
DA-DF	interrupt workspace (R13-R15)
E0-F9	GPL interpreter general workspace (R0-R12)
FA-FF	R13-R15 - GROM pointer, FLAGs, and VDP pointer

3.1.4 VDP RAM Usage

The VDP RAM is used as the primary memory for the interpreter, holding the BASIC program, symbol table, I/O buffer area and the string space, as well as the standard display, character and color table areas. When the expansion memory peripheral is present the program and numeric variable value storage are in the expansion memory, not in the VDP. The discussion here assumes that the memory peripheral is not available and the VDP RAM must be utilized to its fullest. Usage of the expansion memory is described in a later section.

The following table and diagram describe, in general, how the VDP RAM is partitioned for use by the interpreter. More information on the data structures used in these areas can be found in the particular sections of this document which describe them.

Addresses	Use
0000-0300	Screen
0300-0370	Sprite attribute list
0371-03BF	BASIC temporaries
03C0-03FF	Roll-out area
0400-077F	Character tables
0780-07FF	Sprite velocity block
0800-081F	Color tables
0820-08BE	Crunch buffer
08C0-095E	Edit-recall buffer
0960->	Value Stack
dynamic	String Space
dynamic	File attribute blocks
dynamic	BASIC symbol table
dynamic	BASIC line number table
<-3FFF	BASIC program text storage

Figure 3.1.4.1 contains a graphic representation of how the VDP RAM is partitioned for use by the BASIC interpreter.

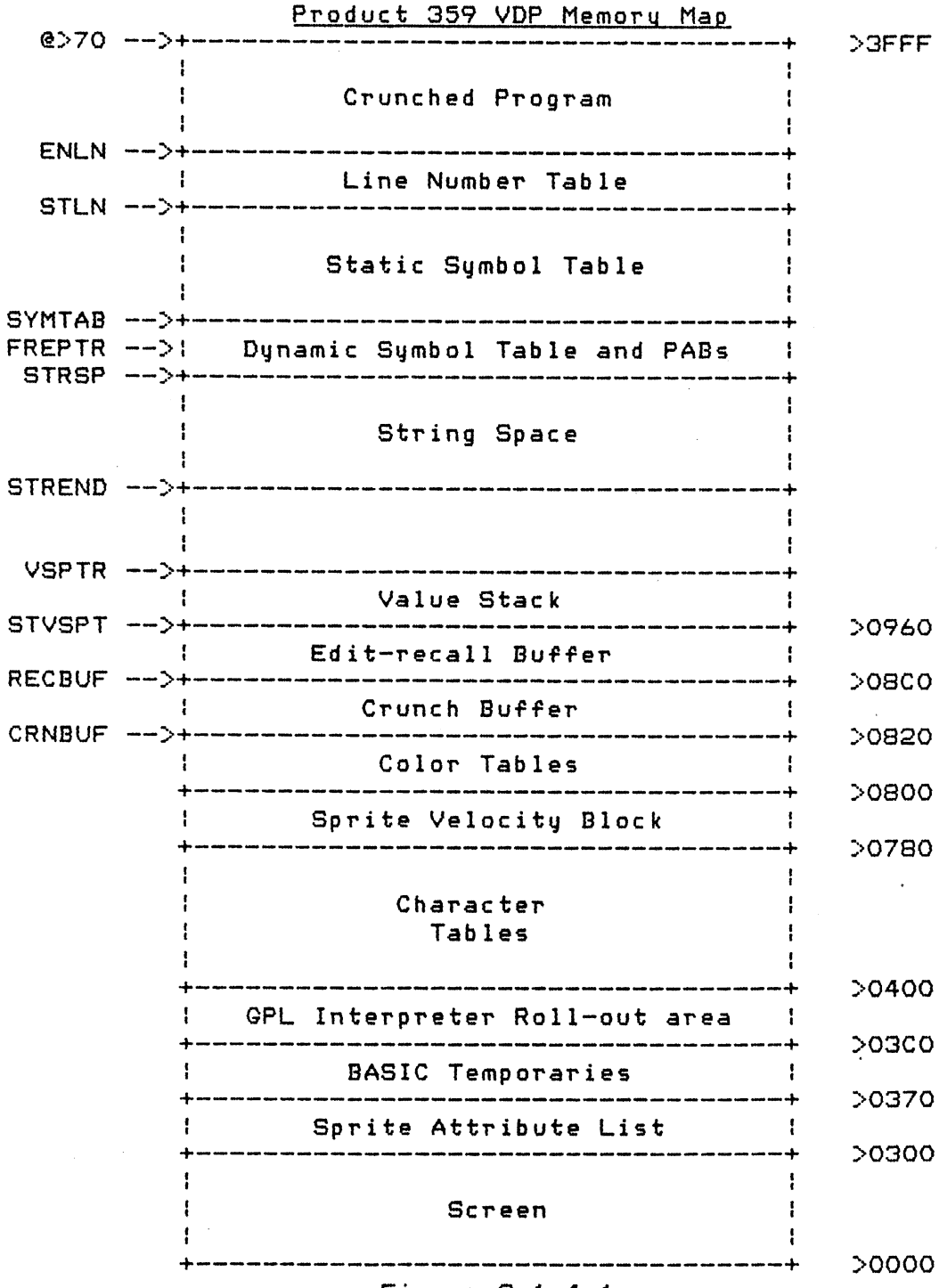


Figure 3.1.4.1

The VDP temporary area is used for maintaining several seldom-used values as well as some things, such as the sound list, which must have space allocated for them in the VDP RAM. A complete list of the VDP temporaries used appears in the

following figure.

EXTENDED BASIC
VDP Addresses

		# of bytes
>370	End of SAL - must always be a >D0	1
>371	LODFLG - flag indicating if auto-boot needed	1
>372	START - line to start execution at	2
>374	unused	2
>376	SYMBOL - permanent symbol table pointer	2
>378	ONECHR - place for one character for CHR$	1
>379	VRMSND - sound list	9
!		
>381		
>382	SPGMPT - saved PGMPTR on break	2
>384	SBUFLV - saved BUFLEV on break	2
>386	SEXTRM - saved EXTRAM on break	2
>388	SAVEVP - saved VSPTR on break	2
>38A	ERRLN - on-error line pointer	2
>38C	BUFSRT - starting screen addr for edit recall	2
>38E	BUFEND - ending screen addr for edit recall	2
>390	CSNTMP - temporary for FAC12 after CSN	2
>392	TABSAV - saved symbol table ptr on break	2
>394	AUTTMP - auto-boot temporary	2
>396	SLSUBP - saved LSUBP on break	2
>398	SFLAG - saved FLAG bits on break	2
>39A	SSTEMP _ subprogram prescan temporary	2
>39C	SSTMP2 _ subprogram prescan temporary	2
>39E	MRGPAB - saved PAB pointer for MERGE	2
>3A0	RNDX2 - random number seed 2	5
>3A5	RNDX1 - random number seed 1	5
>3AA	INPUTP - pointer to input prompt	2
>3AC	ACCVRW - ACCEPT temporaries for VARW,VARA	2
>3AE	ACCVRA - in "TRY AGAIN" case	2
>3B0	VALIDP - pointer to the standard string in VALIDATE	2
>3B2	VALIDL - length of the standard string in VALIDATE	2
>3B4	SIZCCP - SIZE temporary for CCPADR in "try again"	2
>3B6	SIZREC - SIZE temporary for RECLEN in "try again"	1
>3B7	ACCTRY - "try again" flag in ACCEPT	1
>3B8	SIZXPT - save XPT in SIZE in case "try again" case	1
>3B9	SAPROT - PROTECTION flag in SAVE	1
>3BA	CSNTP1 - temporary for FAC10 after CSN	1
>3BB	unused	1
>3BC	OLDTOP - temporary for RELOCA in FLMGR	2
>3BE	NEWTOP - temporary for RELOCA in FLMGR	2

Figure 3.1.4.2

3.1.5 Expansion RAM Usage

The expansion memory, when present, is used for the storage of the BASIC program and the numeric variable values. By using the expansion memory in this manner, larger programs and larger sets of data may be handled by the interpreter. When present,

the expansion memory is utilized in the manner shown in the following diagram.

Product 359 Expansion RAM Memory Map

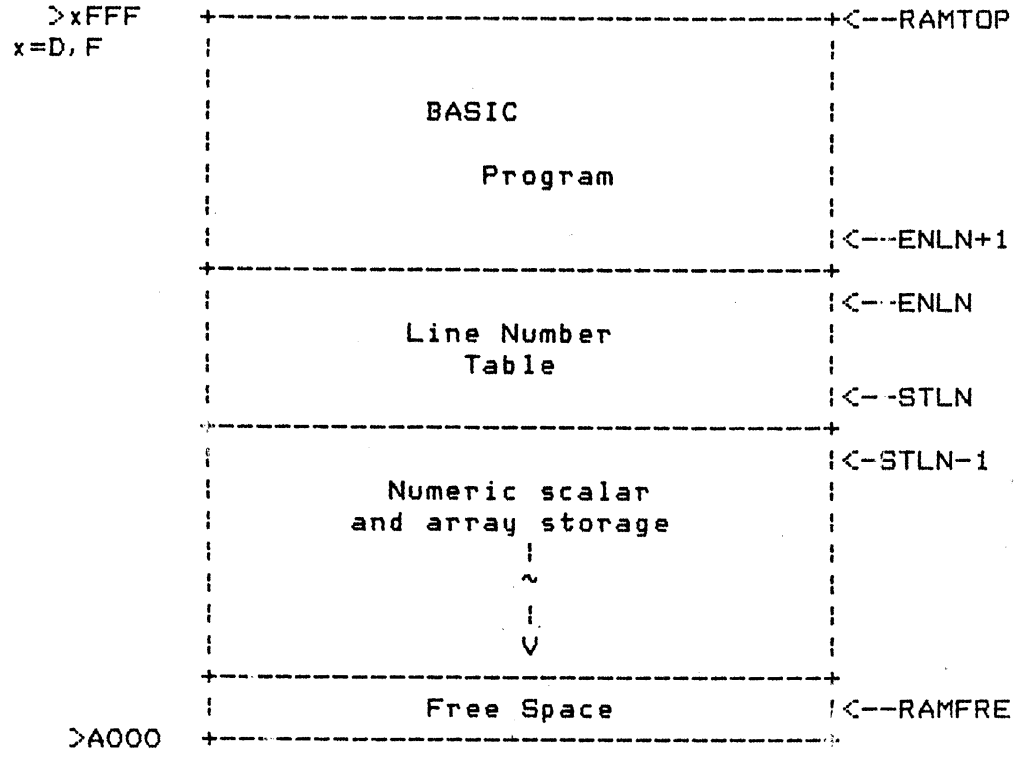


Figure 3.1.5.1

Much more information on how the expansion memory is used and is interfaced to can be found in the Product 359 BASIC Interpreter Expansion RAM Peripheral Support Software Specification. That specification is considered to be a part of this document.

4.0 Interpreter Phases

This section describes the three different phases involved in entering and executing a BASIC statement or program. These phases are: statement / program entry (editing), statement / program prescanning, and statement/program execution. Each of these phases, of course, have several subphases under them and these are also described in detail so that the code can be analyzed and / or modified by persons needing to do so.

The outer-most level of BASIC, commonly referred to as top-level, is the global dispatching element which determines what has been entered from the keyboard (statement, program line, command, etc.) and takes care of dispatching control to the proper handling element. The following is a pseudo-code description of the top-level of BASIC.

```
BEGIN
   REPEAT FOREVER
      READ_INPUT_LINE
      CRUNCH_INPUT_LINE
      SELECT "INPUT LINE"
        WHEN "DIRECT STATEMENT"
            SCAN_NEW_STATEMENT
            PARSE_NEW_STATEMEMT
        WHEN "PROGRAM LINE"
            EDIT_PROGRAM_LINE
        WHEN "COMMAND"
            EXECUTE_COMMAND
        END SELECT
   END REPEAT
END
```

Functionally this is an endless loop, since the BASIC interpreter has no logical end. The interpreter is exited via the execution of a BYE command.

4.1 Editing

The BASIC editor consists of routines to read an input line, crunch a line into tokens where possible, and to handle editing by line number (deletion, addition and replacement). Each of these sections are discussed here.

4.1.1 Input

The line input routine is entitled READLN. This routine takes care of reading a character from the keyboard, displaying the character on the screen and returning to top-level when a line is terminated with a carriage return, and conditionally with up-arrow or down-arrow depending upon the particular mode

which is active when the routine is called.

READLN has three modes of operation when used for accepting BASIC statements and commands. The three modes are: normal line input, auto-num input and edit-mode input.

Briefly, normal line input is when the READLN routine has been called with none of the special modes set. It is merely accepting a line of input. This will become clearer as auto-num input and edit-input are illuminated.

Auto-num input mode is active after the user has entered the NUM or NUMBER command and BASIC is generating sequence, or statement, numbers automatically. Auto-num mode does not allow one to modify the line number generated and this mode is exited by entering a return key without including any other input on the line. If auto-num generates the line number of a statement which already exists in the program the editor goes into edit-mode for that statement as described in the following paragraph. The editor returns to standard auto-num mode when a sequence number is generated which does not appear in the program.

Edit mode is invoked by the user entering the line number of a statement in the program and terminating the line with an up-arrow or down-arrow key. When this occurs the line is displayed on the screen and the cursor is positioned at the first character of the line. At this point all of the line editing features described in the following section become active for that line allowing the user to change a single line in a program without re-entering the entire line. If the user terminates a line in edit mode with an up-arrow, the previous line in the program is displayed on the screen and edit mode is active for that line. If the user terminates a line in edit mode with a down-arrow, the succeeding line in the program is displayed on the screen and may be edited. Edit mode is exitted by terminating a line with the enter key. Also, edit-mode is exited if a down-arrow is entered and the last line of the program is being editted. Similarly, edit-mode is exited if an up-arrow is entered and the first line of the program is being editted.

Whenever a complete line is entered it is immediately saved away, in text form in the line-recall buffer so that it may be recalled when a shift-R is entered as described below. When a line is saved in the recall buffer, the two pointers which point to the beginning of the line and the end of the line, respectively, are saved with the line so that they may be used when the line is recalled.

The READLN routine also contains all of the special line-editing features of this BASIC and they are described in the following section.

4.1.1.1 Line Editing

The line-editor is the lowest level of editing within the line acceptance routines. This editor takes care of all the editing facilities on the line level, i.e. character insertion and deletion, cursor positioning and line clearing.

The line-editor recognizes the shift-S as a back-arrow and moves the cursor back one position on the screen, wrapping around to the previous display-line if at the beginning of a continuation line in the current input-line. Similarly, the line editor recognizes the shift-D as the forward-arrow and advances the cursor one position to the right wrapping around to a continuation line if at the end of a display-line until four lines have been filled.

The line-editor also recognizes two other keys as being special in-line editing keys. The shift-F is recognized as the delete character key and deletes whatever character is located at the cursor's position and shifting the remaining portion of the line which lies to the right of the cursor to the left one position filling in the space left by the deleted character. The shift-G key is recognized as putting the line editor into insert mode. After a shift-G key is recognized all characters subsequently recognized are inserted into the line in the postion immediately preceeding the cursor. Any characters to the right of the cursor which meet the end of the fourth allowable input line are bumped off of the end until the cursor reaches the end of the fourth input line.

The shift-R key, is recognized as the line recall key. When this key is depressed, the current input line is ignored and the most-recently entered line is recalled from the recall buffer and may be modified or simply entered again.

The shift-T key, is recognized as the field clear key and when depresses clears the entire input field between the beginning and ending pointers.

The line-editor also keeps track of any changes in the current input line. If no changes have been made, a flag is set indicating that no changes have been made to the line. This flag is used by the EDIT routines as an indication that the current program line doesn't need to be replaced. This speeds up program editing when the user is only scrolling through the lines in edit-mode.

4.1.2 CRUNCH

The Product 359 BASIC interpreter uses a "crunched" version of the actual BASIC text for direct execution. This enhances the speed with which each program line can be executed since all lines are already pre-processed.

This pre-processing involves the sorting of each item in the BASIC line into one of the following categories :

* BASIC keywords - e.g. FOR, TO, LET etc.. (see Appendix A).
* Unquoted strings - strings within DATA, CALL and IMAGE statements not surrounded by quotes.
* Quoted strings - Any strings surrounded by quotes.
* Line references
* ASCII text - variable names and comments

After the BASIC line has been pre-processed or "crunched", it can be sorted into the current program segment if a line number has been detected at the beginning of the line.

The crunch algorithm is based around the character property table (CPT) and a current crunch mode. Initially when the crunch routine is invoked a normal (non-special) crunch mode is entered where tokens are recognized and variables are recognized and any numerics are crunched in the special numeric format. Whenever a character is scanned by the crunch routine it's character property is looked up in the CPT to determine if the character is, say, an alphabetic or an operator. Characters are accumulated until a character which does not fall into a particular character property is encountered. Keywords in the language are then identified by searching the keyword tables for the accumulated string to see if it is there. If it is the string is replaced by the associated keyword token value, otherwise it is left as a string of characters, which is presumably a variable name. For example, the statement 'A = B' is crunched as ! 41 ! BE ! 42 !. Some keywords cause special crunch cases to occur based upon their meaning and syntactic requirements for execution time. Whenever a special keyword in the language is recognized (e.g. DATA, ::, GOTO, etc.) a special crunch mode is entered. In some cases (GOTO, RUN, GOSUB, etc.) the special mode involves crunching any numerics encountered in the input stream as special line number tokens. In other cases (DATA, IMAGE, etc.) a special mode where characters are crunched as unquoted strings instead of tokens an/or variable names is entered.

The end-of-line is always indicated by putting a zero byte at the end of the crunched line.

4.1.2.1 Specially Crunched Language Elements

There are some elements of a BASIC statement which are crunched specially. In particular, numeric constants, unquoted string constants and line numbers are crunched into special formats.

Each time a keyword in the language is encountered it is translated into its corresponding token (appendix A), e.g. PRINT is translated into a >9C. Whenever a numeric constant is encountered it is converted into the form of:

| >C8 | 1-byte length | ASCII characters of numeric |

An example of this would be the number 3.1415926 would be crunched as:

| C8 | 09 | 33 | 2E | 31 | 34 | 31 | 35 | 39 | 32 | 36 |
| | | 3 | . | 1 | 4 | 1 | 5 | 9 | 2 | 6 |

All numerics which occur in BASIC statements are crunched into this form, including FOR, DATA, PRINT, etc.

Unquoted strings which occur primarily in data statements are crunched in exactly the same manner as numerics, using the same token value, >C8. For example, the BASIC statement 'DATA HELLO' is crunched as:

| 93 | C8 | 05 | 48 | 45 | 4C | 4C | 4F | 00 |
| DATA | | | H | E | L | L | O | |

Other places where unquoted strings occur in this BASIC are the names which are in: CALL, SUB, OLD, SAVE. For example the statement 'CALL CLEAR' is crunched as:

| 9D | C8 | 05 | 43 | 4C | 45 | 41 | 52 | 00 |
| CALL | | | C | L | E | A | R | |

Quoted strings are crunched in identically the same manner as numerics and unquoted strings except that a different identifying token, >C7, is used. Note that when crunching a quoted string double quotes (two quotes in a row) are translated into a single quote and that single quote is included in the string. For example, the quoted string "AB""CD" would be crunched as:

| C7 | 05 | 41 | 42 | 22 | 43 | 44 |
| | | A | B | " | C | D |

One other element of the language is crunched specially, and that is line number references, in such statements as GOTO, BREAK, IF-THEN-ELSE, ON ERROR, etc. Line numbers are crunched as the line number token (>C9) followed by two bytes containing the binary representation of the line number. For example, the statement 'GOTO 1000' is crunched as:

| 86 | C9 | 03 | E8 |

Note that by storing the line number in this manner the GOTO code and the resequence algorithm are greatly simplified over storing the numbers as numerics.

4.1.2.2 Crunching Multi-statement Lines

Crunching input lines with multiple statements on them presents a couple of minor problems. First, the statement separator must be watched for which is a problem because it is the only token in all of BASIC which consists of two other tokens. Normally, without the statement separator, two colons would be crunched as two tokens. This means that several places in the crunch routine a read-ahead must be done when a colon is encountered to determine if it is followed by another colon. A colon can be identified in two ways because of the way the CPT is structured. A colon can be identified by looking for the ASCII colon value itself or by looking for the multi-operator character property since the colon is the only multi-operator which occurs in this BASIC. When two colons in a row are encountered, they are crunched as the single statement separator token and not as two colons.

Another important thing to remember about the crunch algorithm is that whenever a REM statement or a tail remark indicator (!) is encountered, the remainder of the input line is taken as a comment and two colons in a row will not indicate a statement separator, but merely two colons in a comment.

The crunching of data and image statements is handled specially so that any double-colons or exclamation points are included as data items or within the image format, respectively and not as indicating a statement separator or a tail remark.

4.1.3 Program Image

The BASIC program segment consists of a number of crunched program lines. Each line is terminated by a zero byte. For editing purposes, a line length is added at the beginning of each program line.

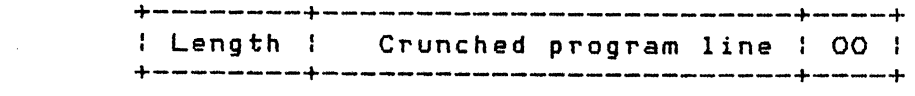


Figure 4.1.3.1 Program line representation

The line number of each program line is stored in a separate "line number table". This table is physically located below (at a lower address) the program text in memory. Each entry in the line number table consists of 4 bytes, containing the line number (2 bytes) and a pointer to the beginning of the corresponding program line in the program text segment (2 bytes). This pointer is not pointing to the length byte of the

program line, but rather to the first item in that line.

This separation of line number table and program text segment was chosen to speed up execution of instructions that reference line numbers (GOTOs, GOSUBs, RESTORE etc.).

The entire program segment is being controlled by three pointers :

* Top of memory pointer, located at location >70 in CPU memory for a VDP based program or at location >84 in CPU memory for an expansion memory based program.
* ENLN(CPU >32) - indicates last location used by the line number table. This pointer is pointed at the least significant byte of the program line pointer.
* STLN(CPU >30) - indicates the lowest or first memory location used by the line number table. This pointer is pointing at the most significant byte of the line number.

The line number entries in the line number table are stored in reverse lexical order, i.e. the highest line number is located at the lowest memory address. Entries in the program text segment are being made in time-order, i.e. the last line entered is stored at the lowest memory address. Entries in the program text segment are therefore <u>not</u> in any logical order. Figure 4.1.3.1 shows the memory image of a program stored in the VDP memory.

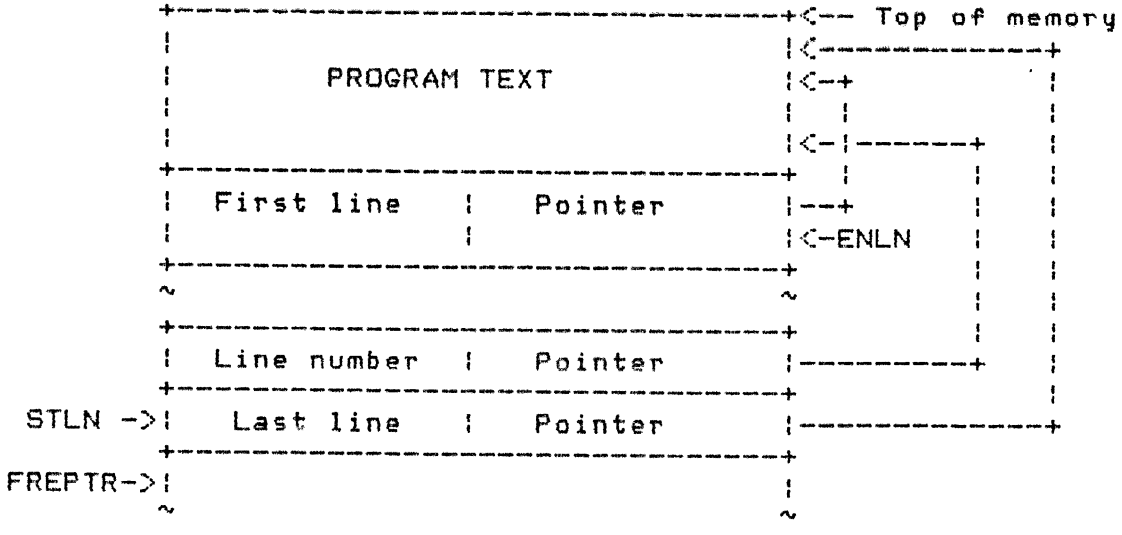


Figure 4.1.3.1 VDP Program memory image

Figure 4.1.3.2 shows the program image of a program stored in the expansion memory.

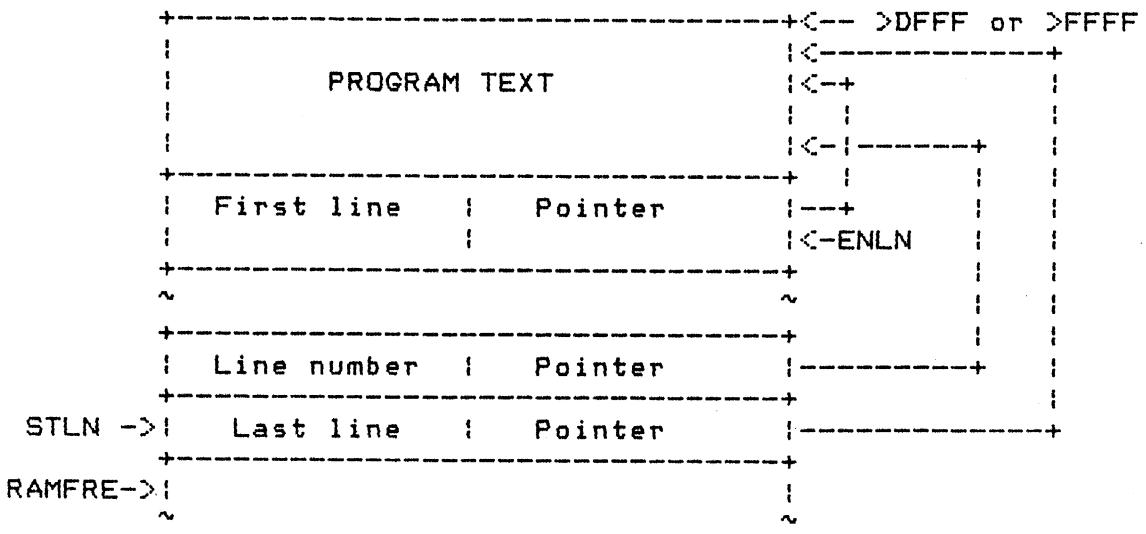


Figure 4.1.3.2 Expansion Memory Program memory image

4.1.3.1 VDP RAM Program Editing

Program editing consists of two separate parts, line deletion, and line insertion. Whenever a line has been entered from the keyboard, a search is made of the line number table to see if a line with the same line number already exists. If it does, the line must be deleted from the text before the new line can be inserted into the program.

In order to delete a line, the text pointer within the line number table is used to get the pointer to the line text. The line's length is then picked up by decrementing the text pointer and getting the length from the VDP RAM. Next, the line text is deleted from the program text area by moving all of the other lines' text that reside at lower memory addresses up in memory to fill the space occupied by the deleted line. The line table is then updated by sequentially going through it and adding the distance moved to the pointers of each program line which moved. The old line number entry is deleted from the table at this time.

In order to insert the new line in the program, the line number table is moved down in memory and the new program line is appended to the lower end of the program text area, just above the line number table. Note that none of the text pointers in the line number table need to be changed as the lines in the program did not move, only the pointers to them. The line number table is now searched to find the correct place to locate the new line number and the new line number, as well as the text pointer are inserted into the table, completing the line insertion.

The following flowchart describes visually how the editor works.

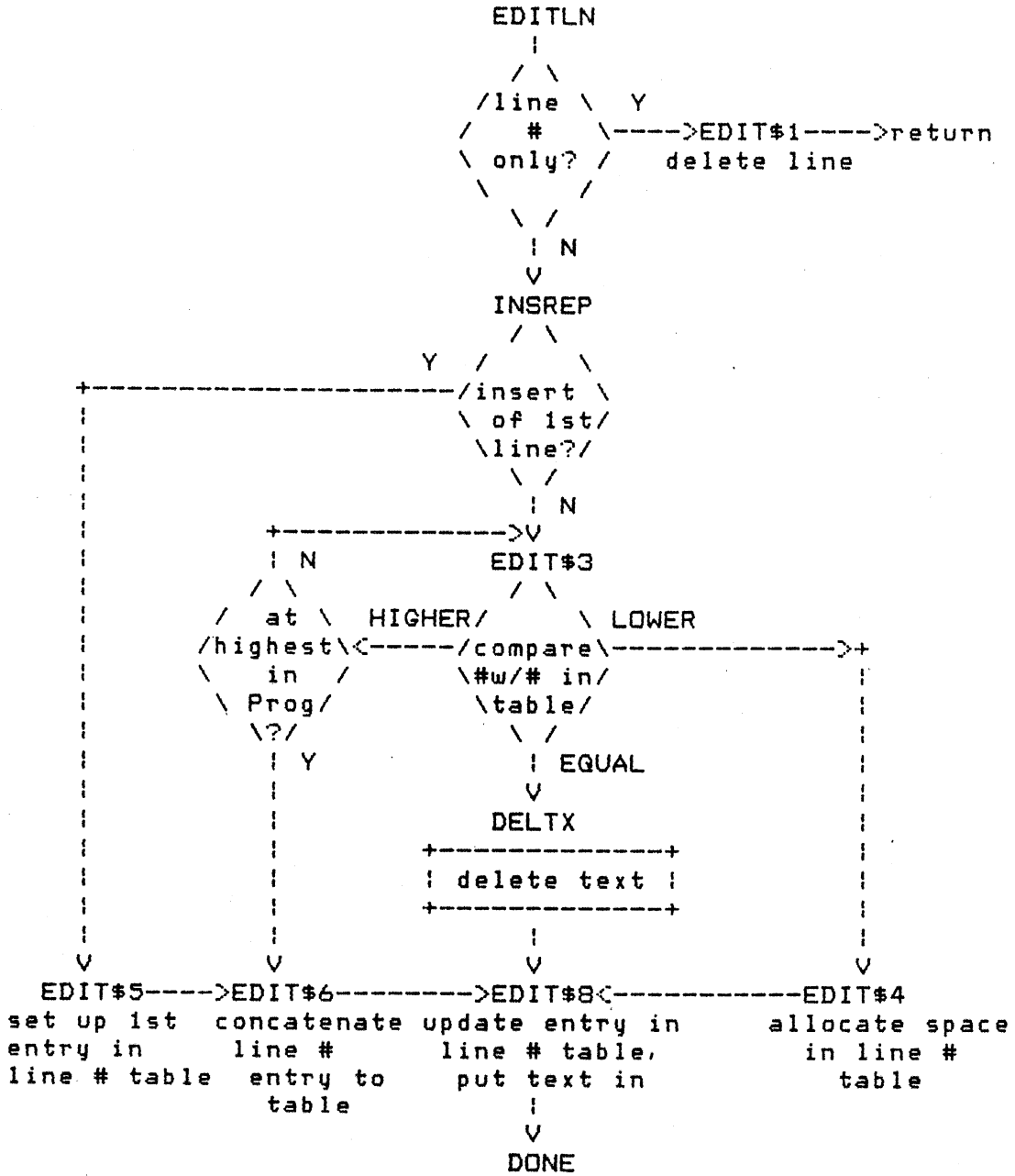


Figure 4.1.3.1.1

4.1.3.2 Expansion-Memory Program Editing

Program editing of a program in the expansion memory is identical to editing a program in the VDP RAM. The same code is utilized for editing a program in the expansion memory except that every place that the memory must be accessed a check is

made to see if the expansion memory is present and if so the access is made to it, rather than to the VDP. More information on editing a program in the expansion RAM can be found in the Product 359 BASIC Interpreter Expansion RAM Support Software Specification.

4.1.4 Auto-num

The auto-num feature is included in the Product 359 BASIC to provide for ease in entering programs into memory. Auto-num consists of two phases within the interpreter. First, is the processing of the number-command and second is the automatic generation of sequence numbers as each new line is entered.

The processing of the number-command involves setting up the two interpreter variables CURLIN and CURINC with the starting line number and the increment for the sequence. First, they are set up with the default values of beginning with line number 100 and incrementing by 10. If either, or both, of the optional arguments are present these values are updated with the supplied value or values. Auto-num then sets the AUTONUM bit in the variable FLAG to indicate that the interpreter is in auto-num mode. Auto-num then returns to top-level and proceeds to generate the line numbers as needed.

Auto-num mode normally takes the current line number value, CURLIN, and adds the current increment value (CURINC) to it to generate the next line number in the sequence. If the next line number to be generated exceeds the implementation defined maximum line number of 32767 then auto-num mode is exited in the same manner as when an empty line is entered to terminate auto-num mode.

In the special case that the line number generated by auto-num mode is the line number of a line in the program in memory, then the line is retrieved from memory and displayed on the screen for editing, just as if edit-mode had been entered. When the line has been editted and the enter key has been pressed, the interpreter returns to standard auto-num mode (enters edit mode again if the next line number generated also matches a line number in the program). Note that when the interpreter is in auto-num mode and edit-mode is entered from that state, the up-arrow and down-arrow keys function exactly as the enter-key, terminating editing of the current line and generating the next line number in sequence, not moving to the previous or next line in the program as in simple edit-mode.

4.1.5 List

In order to be able to look at a program once it has been entered into memory, the list-command is included in the Product 359 BASIC interpreter. The list-command has the ability to list a portion of a program or all of a program, to either the display or a device. Execution of the list-command begins by scanning the list-command for a line-range and for a device name. If a line-range or single number is specified, then two pointers are set up to point into the line number table to the first and last lines to be listed. The pointers can be equal to each other indicating that only one line is to be listed as well as pointing to the first and last lines of a program if the entire program is to be listed.

After the line number range has been found, the list routine checks to see if the listing is to go to the display or to a device. If the listing is to go to the display then the display is initialized for output. If the listing is to go to a device, a dummy PAB is used to open the device and flags are set to indicate that the device is the destination of the listing.

List then goes into a loop, calling the line-listing routine, LLIST, to take an individual line of a program and restore it from its crunched form to a source form and to output it to either the screen or to the device. Special code is utilized to be able to list a program from the expansion memory. After all of the lines to be listed have been put out, a check is made to see if the output went to a device or to the screen. If the listing went to a device, the device is closed. In either case, control returns to top-level. Note that in the case of device output, control returns to top-level by branching to TOPL10 which initializes the symbol table and string space.

4.1.6 Resequence

Execution of the resequence feature of the Product 359 BASIC interpreter is a three-part operation. First, the resequence-command is scanned to pick up the optional starting line number and/or increment. If they are found they are used, otherwise, the defaults of starting with line number 100 and incrementing by 10. The next phase of the resequence operation involves searching all of the text in the program looking for any line number references.

If a reference to a line number is found, the line number table is searched to find it. When it is found, its offset into the table is used to calculate the line number that it will become. When the new line number has been calculated, it is inserted into the program text and the entire process is repeated until all line number references in the program have been found and replaced.

The line number table, itself, is then updated by going sequentially through it placing the new line numbers in the line number portion of each line entry. After the entire operation is completed, control returns to top-level to await the users' next command.

4.2 Prescan

Before a BASIC program is actually executed it must first be scanned in order to generate a symbol table which contains all of the variables used in the program and to generate a subprogram table which contains all of the subprograms referenced in the main program and all of the subprograms it references. The static scanner also builds a symbol table for each of the subprograms referenced so that each subprogram can have its own, local, variables. It also resolves all references to subprograms which occur in the main program and any subprograms it references.

The static scanner also does some error checking which consists mainly of syntactical checks and for consistent use of all symbols. The scanner checks to make sure that each FOR statement has a corresponding NEXT statement, that INPUT, LINPUT, DATA, GOTO, GOSUB, DEF, RETURN, IMAGE and OPTION BASE statements are not used imperatively. It also insures that SUB, SUBEND, NEXT, FOR, OPTION, DIM and DEF do not appear inside a THEN or ELSE clause of an IF statement. THEN and ELSE must occur in an IF statement.

In order to lessen the time spent between when a run command is entered and execution actually begins, the static scanner also looks for a DATA statement in order to initialize the two pointers, DATA and LNBUF, needed by the read statement to keep track of the current data position within the program.

After a program or imperative statement has been properly scanned control flows to the execution portion of BASIC to execute the program or statement.

4.2.1 Symbol Table

In order to be able to use variables in a BASIC program memory must be set aside to describe each of the variables and to store each variable's value. In order to do this, this BASIC prescans each program line and builds a symbol table containing an entry for each symbol (variable) used in the program. A standard format is used for all symbol table entries. This format is shown in figure 4.2.1.1 below.

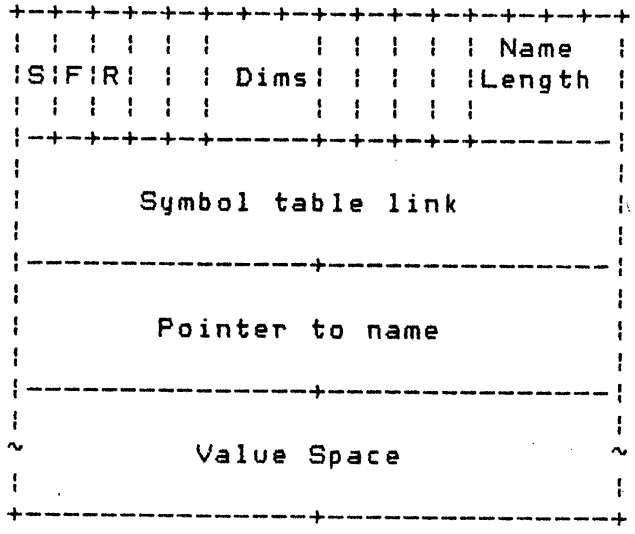


Figure 4.2.1.1

The following meanings have been attached to the bytes contained in a symbol table entry.

WORD 1
- S - String flag - set if a string variable else reset
- F - User-defined function flag - set if user-defined function else reset
- R - Share flag, indicating a shared symbol(in subprogram calls) - set if shared, reset at all other times.
- Dims - Number of dimensions or function parameters - legal values 0 - 7
- Name length - Length of symbol's name - legal values 0 - 15

WORD 2
- Symbol table link - link to next entry in table or zero if end of table

WORD 3
- Pointer to name - pointer to an occurrence of the symbol's name

WORD 4+
- Value space - symbol's value space which contains:
  - * Dimension information if array entry, 2-byte maxima per dimension
  - * 8-byte entry(ies) for numerics;
  - * 2-byte pointer to value in expansion RAM for numerics;
  - * 2-byte entry(ies) for strings;
  - * 2-byte pointer to definition for user-defined functions;
  - * 2-byte pointer if a shared symbol.

4.2.2 Subprogram's Symbol Tables

A subprogram's symbol table is virtually identical to the main program's symbol table, except that the new bit in the header is used. The new bit, the shared bit is set when a formal parameter is sharing a value with an actual.

In the case of an formal being shared the shared bit is set and the value space for that entry contains the address of the value space of the actual it is sharing the value with. The following diagram shows this condition.

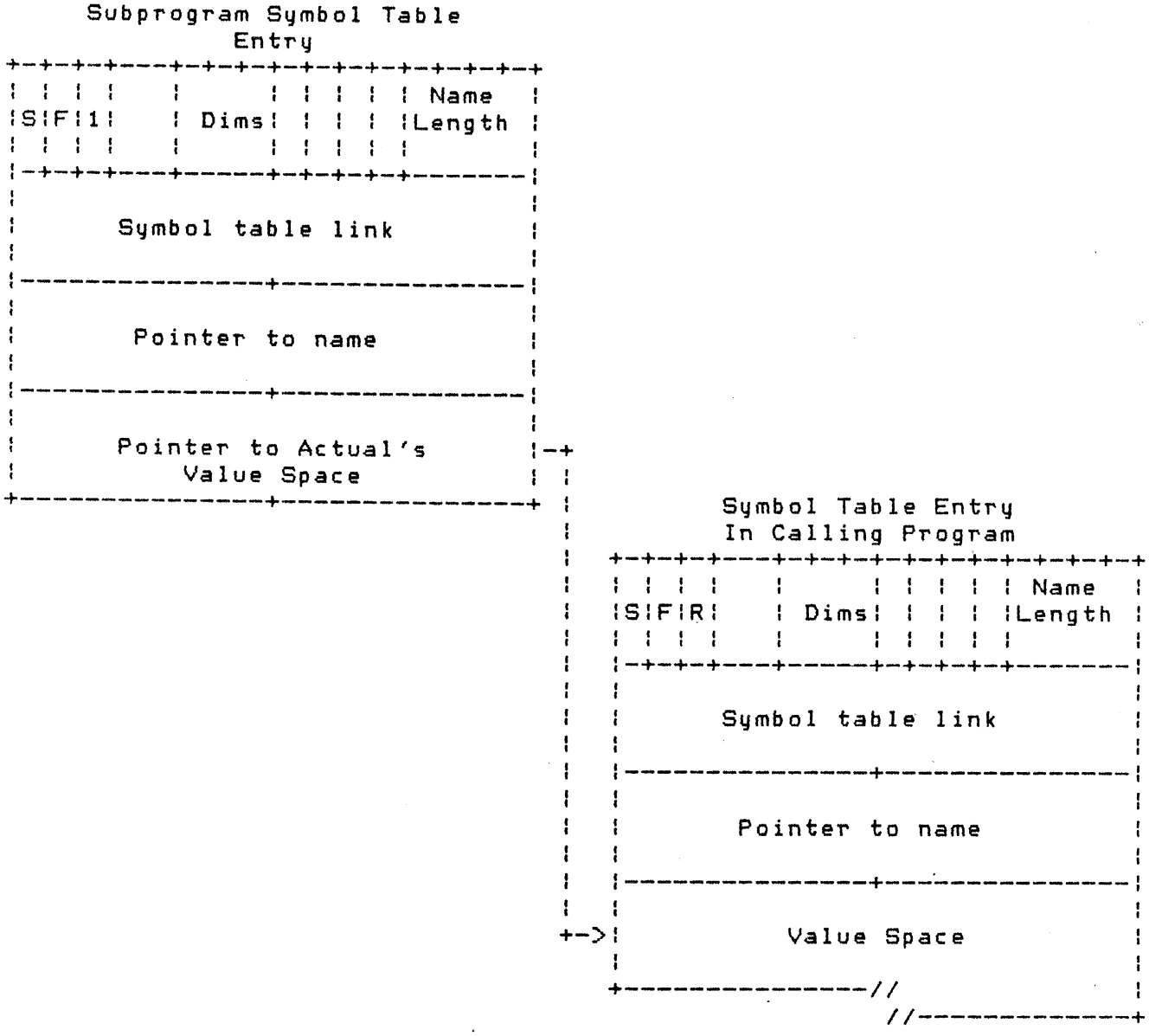


Figure 4.2.2.1

Note that in the case of an array being shared, the pointer into the value space points directly at the dimension maxima information. For this reason redimensioning of arrays is not possible. For more information on subprograms see the Product 359 BASIC Subprogram Specification.

4.2.3 Subprogram Name Symbol Table

This section deals with the special symbol table generated for subprogram headers. This table contains only entries for

subprograms that are defined either internally, or contained externally, as Graphics Language subprograms. BASIC subprogram entries and assembly (GPL) language entries contain different information due to the fact that assembly language routines must be accessed by an absolute address and do not have BASIC symbol table entries passed to them by the interpreter. BASIC subprograms are loaded via the internal relative addressing structure of BASIC for internal subprograms and must have provisions made for the passing of parameters. Assembly Language (GPL) subprograms handle this themselves.

The subprogram symbol table has essentially the same format as a variable symbol table so that common routines can be used in constructing them and searching them. Please note that certain flag bits in the subprogram symbol table entries have a completely <u>different</u> meaning than their counterparts in the variable symbol tables. No conflicts arise from this condition because separate code is used for interpreting the flag bits in the header.

Figure 4.2.3.1 is a diagram of the subprogram table entry format for BASIC language subprograms.

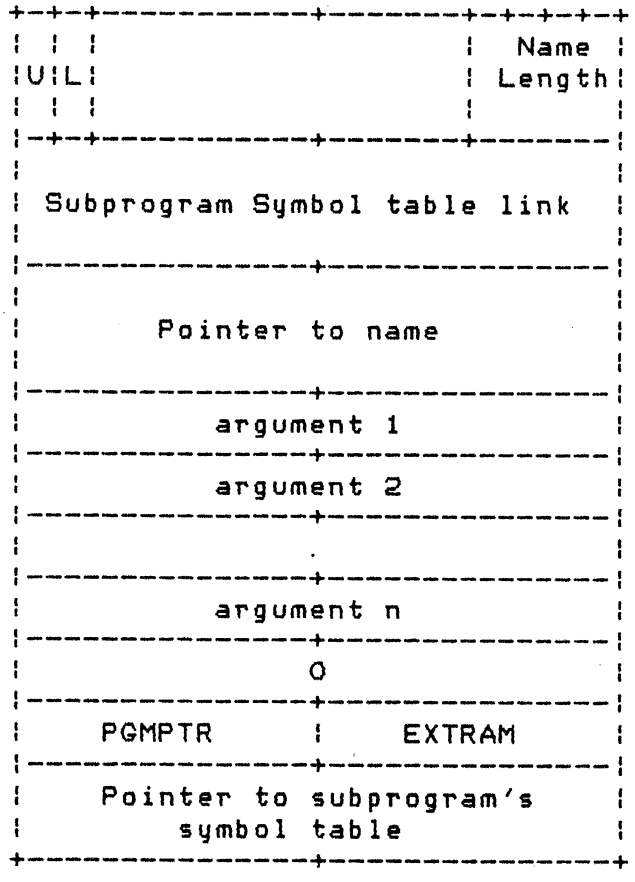


Figure 4.2.3.1

WORD 1
U - In-use flag: set if subprogram in use, reset if not
L - Language flag: reset since BASIC language
Name length - Length of symbol's name: range - 0-15

WORD 2
Subprogram symbol table link - link to next entry in table or zero (0) if end of table

WORD 3
Pointer to name - pointer to an occurrence of the symbol's name

WORD 4+
argument 1...argument n - pointers to subprogram symbol table entries for formal arguments
0 - Signifies end of arguments and beginning of environment information
PGMPTR - Pointer into the sub-statement of the subprogram
EXTRAM - Pointer into line number table to line that contains the sub-statement of the subprogram
Pointer to subprogram's symbol table - pointer to first entry in the subprogram's symbol table or zero (0) if table empty

Figure 4.2.3.2 is a diagram of the subprogram table entry format for Assembly (Graphics) Language subprograms.

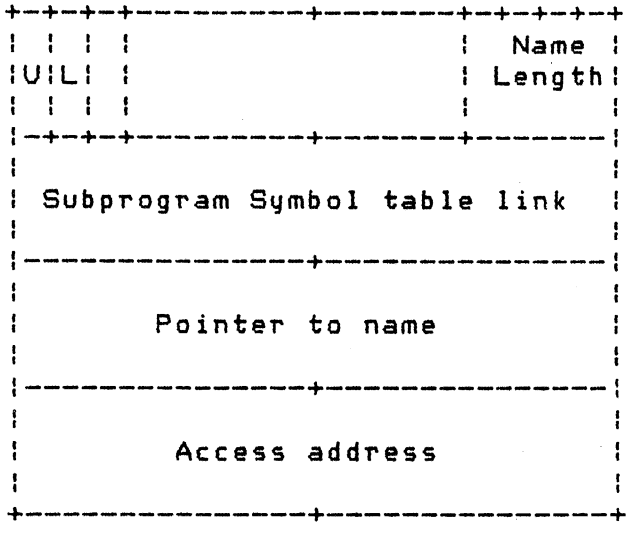


Figure 4.2.3.2

WORD 1
U - In use flag: set if subprogram in use, reset if not
L - Language flag: set, since assembly language
Name length - Length of symbol's name: range - 0-15

WORD 2
Subprogram symbol table link - link to next entry in table or zero (0) if end of table

WORD 3
Pointer to name - pointer to an occurrence of the symbol's name

WORD 4
Access address - absolute address of the subprogram

The in-use flag is set whenever a subprogram is currently active. Whenever a subprogram is invoked the in-use flag is checked and if it is set an error occurs due to an attempted recursive invocation, either directly or indirectly. If the in-use flag is not set at that point it is set before control is turned over to the subprogram. The in-use flag is reset whenever a subprogram is exited. For more information on subprograms see the Product 359 BASIC Subprogram Specification.

4.3 Execution

A BASIC program or imperative statement is executed by the use of the XML EXECG command in GPL. When the command is executed the Assembly Language portion of BASIC takes care of the parsing of each statement until the program or statement has completed or an error has occurred. Generally speaking, every keyword in BASIC is contained in one or more tables and, when it is encountered in a statement, a keyword-specific section of code is used to interpret a particular keyword's meaning. The keyword table is ordered in such a way (see Appendix A) that a strict precedence is set up which keeps the interpreter from allowing any illegal statements to be executed. Whenever a variable name is encountered in a program it is looked up in the symbol table which was built the static scanner and its value is picked up or the variable is prepped for assignment, depending upon its usage in any particular case. The interpreter continues executing until the end of the statement is reached. Subsequent statements in a program are executed unless an error or breakpoint halts execution prematurely.

4.3.1 EXEC

The execution of a BASIC statement or program begins with the acceptance of either the imperative statement or the RUN-command. If a RUN-command was entered, the line number table pointer is set up to either the first line in the program or to the line specified in the RUN command. This pointer is left in the VDP variable, START, and is preserved throughout the prescanning of the program. If an imperative statement was entered, it is prescanned and then immediately executed.

In order to execute a BASIC statement or program, control flows to the beginning of the EXEC assembly of the interpreter. If a program is to be executed, the saved line table pointer in START is moved to EXTRAM to allow the program to begin execution at some line other than the first one. If a single statement is to be executed the text pointer is set to point into the crunch buffer so that that statement can be executed out of there

instead of out of the program.

The XML EXECG command in GPL is used to execute one or more lines of BASIC. The location of the EXEC handler is found in the appropriate XML table and dispatches control to EXECG in the PARSE assembly of BASIC. EXECG sets up the necessary registers for use by the assembly language portions of BASIC (R8 and R9) and then checks to see if a single BASIC statement (imperative mode) or a BASIC program is to be executed.

If a single statement is to be executed, the return address is saved and the first token in the statement is picked up and the statement table (STMTTB) is searched for the token. If the token is not in the table or is a variable name, the statement is executed from there by using the address of the particular statement handler to get to the appropriate statement-specific code.

If a program is to be executed, the interrupt level is set to 3 to allow the 9900 to handle any interrupts that it may have received. The interrupt level is then set to 0 to not allow any interrupts to occur while in the process of executing the 9900 code portion of BASIC. The next statement to be executed is picked up by getting the pointer to it from the line number table. The statement's first token is then picked up and from there the statement is handled in the same manner as the single statement described above.

After a statement has been executed successfully (errors abort execution as soon as they are detected), control returns to the EXEC code and the check for imperative-mode is made again. If in imperative-mode, EXEC is exited and control returns to top-level by returning to the GPL interpreter. If in program-mode, the line table pointer is updated and if not at the end of the program the next statement is executed in the same manner as described above. When the end of the program has been reached, control returns to top-level in the same manner as when a single statement has been executed.

4.3.1.1 Statements

Each BASIC statement has unique code within the interpreter to handle the peculiarities associated with that particular statement. When a statement is to be executed its first token is picked up from the statement text. A valid BASIC token is expected. If a valid token is not received, the statement is assumed to be an assignment statement which does not use the optional LET and control immediately flows to the symbol code (PSYM) which searches the symbol table for the name at the beginning of the statement. If a valid BASIC token appears at the beginning of the statement, its value is checked to see if it falls within the bounds (>80 - >AA) of the valid statement

tokens (see Appendix A for token values). If it does not, an attempt is being made to execute an illegal statement and an error occurs. If the token falls within the legal statement token range the address of the statement handler is looked up in the statement table (STMTTB) and control flows to the appropriate code.

If the most-significant bit of the statement address looked up in the statement table is set to a one, the code to handle the statement resides in the GROM portion of the interpreter and the address picked up is actually an offset into the table, NUDTAB, in the GROM code. The most-significant bit is then reset, and the address within NUDTAB is computed and control returns to the GPL portion of BASIC to execute that statement. Note that because of this condition, an XML PARSE command in GPL cannot occur in any GROMs except GROMs 4 through 7. For this reason, the Product 359 BASIC has been arranged so that all of the parses that need to be done from GPL code have been placed in GROMs 4 and 5.

Note that some of the tokens in the legal statement range are not legal statements (e.g. THEN, ELSE, TO, SUB, etc.) and the address picked up out of the statement table is that of a routine to handle an error so that a statement cannot begin with one of the tokens in the legal statement range which is not actually a BASIC statement.

Each statement then has the responsibility of advancing the text pointer through the statement to execute the statement. When the statement has been completed, the statement handler returns control to the routine, CONT, which verifies that the end of the statement has been reached (CONT is also used in conjunction with the PARSE statement described in the following section) and returns control to EXEC to either execute the next statement on the line, the next line in the program or to return to top-level.

4.3.2 PARSE

The parsing algorithm used in the Product 359 BASIC interpreter is based upon the paper by Pratt listed in the applicable documents.

Pratt's approach associates the semantics of each token with the token via a body of code unique for each token. Some tokens may logically have two such pieces of code, which he calls the null designator (NUD) and the left designator (LED). An example is the token '-' which may be both unary and binary. The unary meaning is described in the NUD for '-', while the binary meaning is described in the LED. Each token is associated with a left binding power (LBP) which is used to supply the precedence for a set of tokens. As the parser algorithm is called initially and subsequently by recursion, a current right

binding power (RBP) is maintained. Each call to the parser supplies a RBP, which in effect tells the parser how far down the input it must process before returning with a value. The parser returns one value on the value stack each time it is called. The subroutine stack is used for return addresses and to communicate the current RBP. Simply stated, the algorithm for parsing is as follows:

```
PARSE(RBP)  Returns value on value stack
     advance token pointer;
     execute NUD for entry token;
     push value from nud onto stack;
     while RBP < LBP
          advance token pointer;
          execute LED for entry token;
          push value from led onto stack;
     end while;
     return;
```

In order to use this algorithm, a NUD, LED and left binding power (LBP) have been assigned by the token values used and by the tables in the PARSE assembly of BASIC.

4.3.2.1 Precedence

The precedence of the BASIC tokens have been assigned in order to allow the use of the algorithm described in the previous section. The tokens were assigned so that the statement tokens are grouped together, the NUD tokens are grouped together and the LED tokens are grouped together, as much as possible. This allows a global range checking to make sure tokens appear in the correct areas of the tables so that a semantic meaning may be attached to each token. The lowest valued tokens are those which appear in the statement tables and which must be either the first token to appear in a statement (FOR, LET, PRINT, etc.) or must be used in conjunction with one of these tokens (THEN, TO, STEP, etc.). The remainder of the tokens appear in the NUD tables with a small portion of these also appearing in the LED tables (+, -, =, etc.).

4.3.2.2 NUDs and LEDs

All of the tokens in BASIC have a unique piece of code to handle them, with some of the tokens having two such pieces of code. These are known as the null designator (NUD) and the left designator (LED) codes. The NUD code is the code denoted by a token without a preceeding expression (e.g. +3, -4). Thus, for the unary use of '+' and '-' the semantic code for each is placed in the tables as a NUD routine. In contrast, the code for

the binary operators of '+' and '-' are placed in the tables as LED routines (e.g. 3+4, 5-6)

As can be seen, there naturally are more NUDs in BASIC than there are LEDs. The entire LED table includes the '&', '=', '<', '>', '+', '-', '*', '/', and the '^' operators. The NUD tables include all of the rest of the tokens in BASIC which are not contained in the statement table. This includes such tokens as '(', '+', '-', SIN, COS, SEG$, the numeric constant token and the string constant token.

4.3.2.3 CONTINUE

Based upon the algorithm described in the preceeding sections on how the parser in this BASIC works, it can be seen that a routine to check the precedence of the left binding powers (LBPs) and the right binding powers (RBPs) is needed in order to tell the parser how far down the statement to continue before returning a value. The continuation routine, CONT, serves this purpose. Its sole purpose is to compare the current precedence with the previous precedence to determine whether the statement has been parsed far enough or of if more parsing at the current level must be done. The continuation routine is accessed from GPL by the execution of the XML CONT command and is accessed from assembly language by the execution of a B @CONT instruction.

4.3.2.4 Multi-statement Lines' Execution

In order to support multi-statement lines, some changes have to be made to the front-end and tail-end code of the parser, specifically in EXECG and in CONT. EXECG must be modified to set up the top-level right binding power (RBP) to be equal to the tail-remark token so that the parser will treat a current token value of >83 (tail-remark token) or lower as an end of statement. When a token which meets that criterion is found it is known to be one of three possible tokens: (1) a tail-remark(>83), (2) a statement-separator(>82), or (3) an end-of-line(>00). The return address which is pushed on the subroutine stack when a statement begins execution causes control to go to the EXRTN label to end a statement. The code at this location must check to see if a statement-separator has been detected and if it has execution must continue with the following statement. If a statement-separator is not the current token, a tail-remark or an end-of-line token has been reached and control flows to the code which checks to see if there are more lines in the program to be executed and, if so, they are executed in the same manner as the line just completed. If the end of the program has been reached or an imperative statement was executed, control returns to the top-level of BASIC to

accept the next input from the user.

Additional code must be written so that non-executable statements, such as option-base-statements and dim-statements, can be properly skipped over by the parser. When one of these statements is detected, a scan of the statement is made to see if another statement follows it on that line. If there is another statement it must be executed. The code at NUDEND is designed to take care of this situation. It scans a line beginning at the current token and continues until a statement-separator, tail-remark or an end-of-line is detected. When one of these is detected, control returns to EXECG to determine exactly how to handle the token.

Whenever a rem-statement, tail-remark, data-statement or image-statement is encountered in a program execution flows to code that immediately ends the current line and returns control to EXECG to proceed to the next line. This means that these statements are completely ignored and are not scanned looking for a statement-separator as dim and option-statements are scanned as described above.

4.3.3 Data Structures

This section describes the two outstanding data structures which are used by BASIC during execution time which have not been described previously. These data structures are the value stack and the string space.

The value stack is very intimately tied with the parsing operations described in the previous section. It is the main means of accumulating and passing data within a particular statement evaluation and the main means of maintaining information between any statements which dictate that information be maintained (FOR, GOSUB, etc.).

The string space comes into play whenever a string is used within a BASIC statement or program. All strings are copied into the string space as they are used and the string space is maintained during the entire time a BASIC statement and/or program is being executed. This data structure allows for the feature of variable-length strings in BASIC, from 0 to 255 characters.

4.3.3.1 Value Stack

Most statements in BASIC are executed at a particular time in a program and completed with control never returning to them (except if within a loop). There are a few statements which cause control to pass to some other portion of a program and then require a return to the original execution point. These

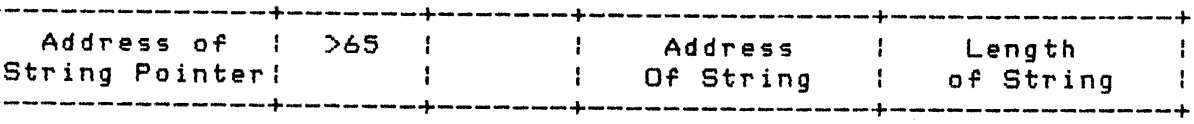
statements require that certain information be kept around even when not executing that particular statement. The FOR/NEXT, GOSUB and CALL-statements and references to user-defined functions fall into this category. Each requires return information of some sort, as well as other information peculiar to each. To maintain this information, one or more 8-byte entries are pushed onto the value stack.

Each stack entry has an identification byte which is the third byte (FAC+2 or RAM(2(VSPTR)) of the top entry. The ID is kept so that stack searches may be made (in the case of FOR/NEXT) and to prevent the user from doing something illegal and getting garbage on the stack. The third byte of the FAC/stack entry was chosen because it is the first byte of a floating-point number which cannot have a value higher than 99 (>63). The first two bytes of the entry can have a value higher than 99 (>63) because, for negative numbers, the first two bytes of the number are negated. Also, when an integer variable type is added to a future BASIC the first two bytes of the entry will contain the value and the third byte can be used as an ID to indicate that the FAC/stack entry contains an integer (ID >64 was reserved for this purpose).

Other common entries on the stack are numeric entries and string entries. String entries, in addition to the entries alluded to above, are also special. Numerics appear on the stack as 8-byte radix 100 numerals (this is the reason why a stack/FAC entry is 8-bytes wide).

The stack is built in the VDP memory from >960 and may grow as high in memory as needed or until it is about to collide with the string space. The operations, VPUSH and VPOP, move 8-byte entries from the FAC to the stack and vice-versa. VPUSH and VPOP work specially on string entries (see section 4.3.3.2). When the stack and string space get close enough that they would collide, either by pushing an entry on the stack or by allocating a new string, a garbage collection is performed (see section 4.3.3.2).

The stack/FAC entry for a string looks like:

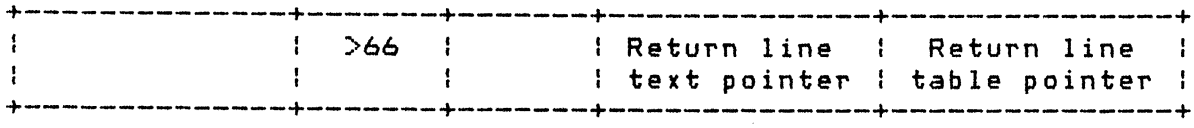


String
Figure 4.3.3.1.1

FAC,FAC+1 - Address of where the pointer to the string came from. Address of SREF (>001C) if a temporary or the address of the symbol table entry if a permanent.
FAC+2 - >65 is the string identification byte
FAC+3 - unused
FAC+4,FAC+5 - Address of the first character of the string

FAC+6,FAC+7 - Length of the string (actually in FAC+7 and FAC+6=0)

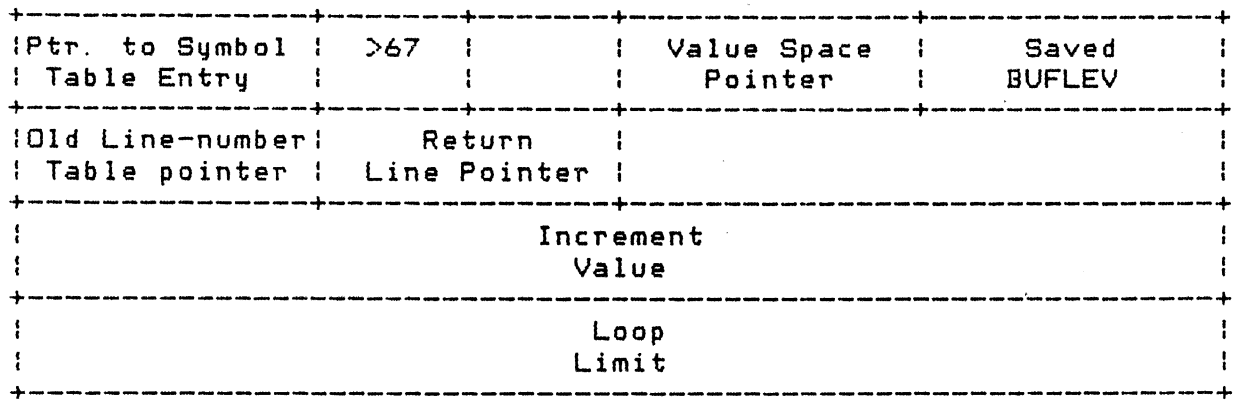
The stack entry for a GOSUB statement looks like:



GOSUB
Figure 4.3.3.1.2

FAC,FAC+1 - unused
FAC+2 - >66 is the GOSUB identification byte
FAC+3 - unused
FAC+4,FAC+5 - Return line text pointer (PGMPTR)
FAC+6,FAC+7 - Return line-number table pointer (EXTRAM)

The stack entry for a FOR statement looks like:



FOR
Figure 4.3.3.1.3

Entry 1
FAC,FAC+1 - Pointer to indeces' symbol table entry
FAC+2 - >67 is the FOR identification byte
FAC+3 - unused
FAC+4,FAC+5 - Pointer to indeces' value space
FAC+6,FAC+7 - Current Crunch-buffer level

Entry 2
FAC,FAC+1 - Line number table pointer to FOR line
FAC+2,FAC+3 - Pointer to within the FOR-statement's line
FAC+4-FAC+7 - unused

Entry 3
FAC-FAC+7 - Value of increment for index variable

Entry 4
FAC-FAC+7 - Value of index limit

The stack entry for a user-defined function looks like:

Return Line Pointer	>68 >70	Function Type	Old Symbol Table Pointer	Old Free Space Pointer

User-defined function
Figure 4.3.3.1.4

FAC,FAC+1 - Return line pointer (PGMPTR)
FAC+2 - >68 is the user-defined function identification byte.
>70 when entering the parameter into the symbol table.
FAC+3 - Function type, >00=numeric, >80=string
FAC+4,FAC+5 - Return symbol table pointer (SYMTAB)
FAC+6,FAC+7 - Return free space pointer (FREPTR)

The stack entry for an error block looks like:

Error Code	Sever-ity	>69	LUNO/ Exec flag	Error Line Table Pointer	Error Line Pointer

Error
Figure 4.3.3.1.5

FAC - Error code
FAC+1 - Severity code of the error
FAC+2 - >69 is the error identification byte.
FAC+3 - LUNO number if device error else 0 if execution error
FAC+4,FAC+5 - Error line table pointer of the line where the error occurred
FAC+6,FAC+7 - Error line pointer to the beginning of the statement where the error occurred

The stack block for a BASIC subprogram call looks like:

Subprogram Name Symbol Table Pointer	>6A	Warning /Break Bits		LSUBP
Return PGMPTR	Return EXTRAM	Return SYMTAB	Return RAM(SYMBOL)	

CALL
Figure 4.3.3.6

Entry 1
- FAC,FAC+1 - Subprogram Name Symbol Table Pointer - The pointer to the subprogram name symbol table entry for this particular subprogram.
- FAC+2 - >6A is the subprogram identification byte
- FAC+3 - Warning/Break Bits - The current on-warning condition (PRINT,NEXT,STOP) and current on-break condition (NEXT, STOP)
- FAC4,FAC+5 - unused
- FAC+6,FAC+7 - LSUBP - Pointer to the last subprogram block on the stack or 0 if there is not one currently on the stack.

Entry 2
- FAC,FAC+1 - Return PGMPTR - Pointer to where program execution is to resume after the subprogram is finished executing.
- FAC+2,FAC+3 - Return EXTRAM - Pointer into the line number table of the line which invoked the subprogram.
- FAC+4,FAC+5 - Return SYMTAB - Symbol table pointer that was active at the time of the subprogram invocation.
- FAC+6,FAC+7 - RAM(SYMBOL) - Symbol table pointer for the first entry in the current symbol table where user-defined function parameters may be attached (usually the same as SYMTAB)

4.3.3.2 String Space

The string management scheme is a very complicated one which allows strings of up to, and including, 255 characters. Strings are located in the dynamic memory area between the symbol table and the value stack. A string has exactly one owner any point in the flow of a user's program. Temporary strings, such as the string "HELLO", in the statement, PRINT "HELLO", are copied into the string space from the program and are left, marked as unused, to be reclaimed when memory is full and a garbage collection must be performed.

User-accessible memory in the VDP RAM between addresses

>960 and >3FFF is used for the storage of the user's program, the line number table, the symbol table and Peripheral Access Blocks (PABs) for file and device operations, the dynamic string space and the value stack. Memory usage is, generally speaking, layed out as follows:

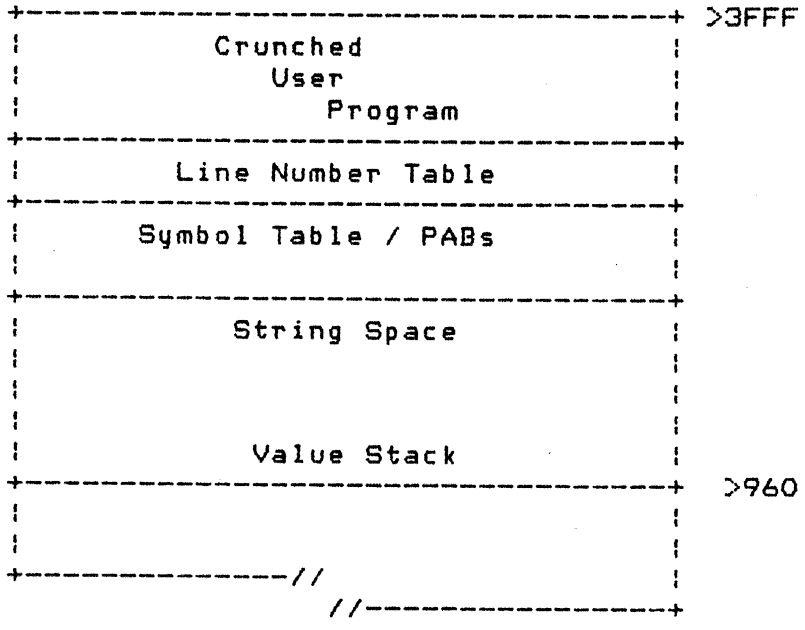


Figure 4.3.3.2.1

The complications involved with managing the memory can be realized by seeing that there are five independent data structures residing in the same memory area. No conflicts arrise between the crunched program and the string space as the program and its associated line number table always reside in the highest addresses possible and whenever the program is modified by the user the symbol table and string space are destroyed.

The symbol table and PABs present a problem in that there can be, and probably are, strings in the string space when a symbol table or PAB entry is allocated. Symbol table entries and PAB's occupy contiguous memory and when an entry is allocated the entire string space is moved down (lower address) in memory to give the needed space. The routine, MEMCHK, described below, takes care of this problem.

As can be seen, the string space can collide with the value stack. When this is about to occur a garbage collection is performed to reclaim any memory that is not being used. Unused memory can be generated in several ways. First, and foremost, by temporary strings as described above. Second, by the closing of a file and the consequent deletion of the associated PAB. Third, by the elimination of a symbol table entry which is no longer

needed (e.g., an entry for a user-defined function parameter after the function has been evaluated).

Following is a detailed description of how strings are created, used, and destroyed in the course of executing a BASIC program.

First, the string, itself, is copied into the string space with a leading and trailing count of the number of characters in it and a 2-byte back pointer which is used for the purpose of garbage collection.

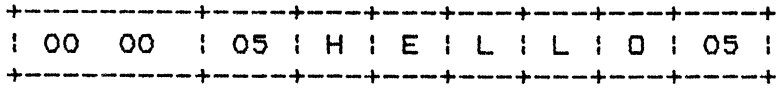


Figure 4.3.3.2.2

In the above example, the backpointer is set to zero, indicating that this is a temporary string and can be reclaimed during a garbage collection. The leading and trailing length bytes (hexadecimal) are used as the length of the string (leading) by the string functions, such as LEN, and by the garbage collector (trailing), as will be explained below. The string is, of course, stored in its ASCII representation.

If the string is assigned to a variable, then the following is a graphic representation of the final result.

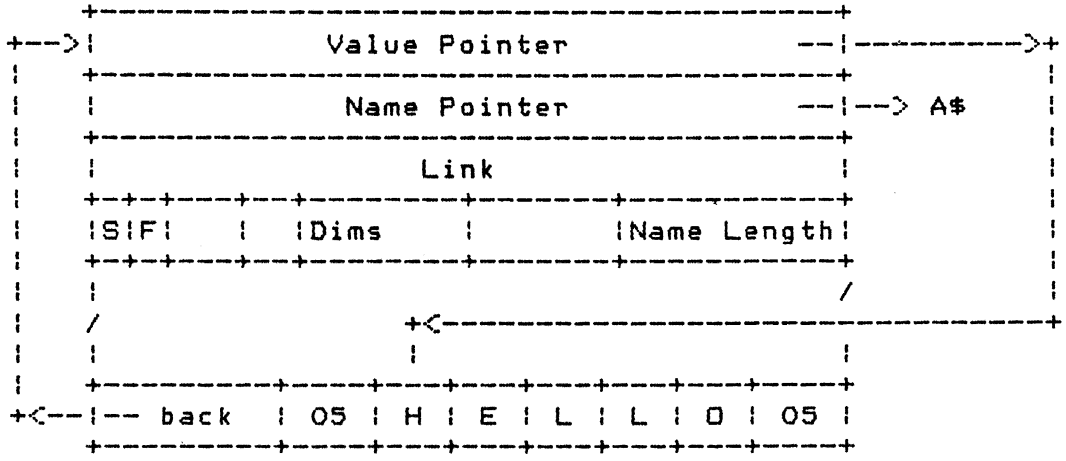


Figure 4.3.3.2.3

The value pointer and backpointer are 2-byte addresses. The value pointer points to the first character of the string and the backpointer points to the address of the value pointer, forming a loop.

When, for example, a DISPLAY A$ statement is executed the pointer is picked up from the symbol table, one is subtracted from it to get the length, 5, and the next five bytes (characters) are printed. If an A$=A$ statement is executed, a completely <u>new</u> string is created, the old one is freed and the new one is assigned to A$. Using this example, memory would now look like:

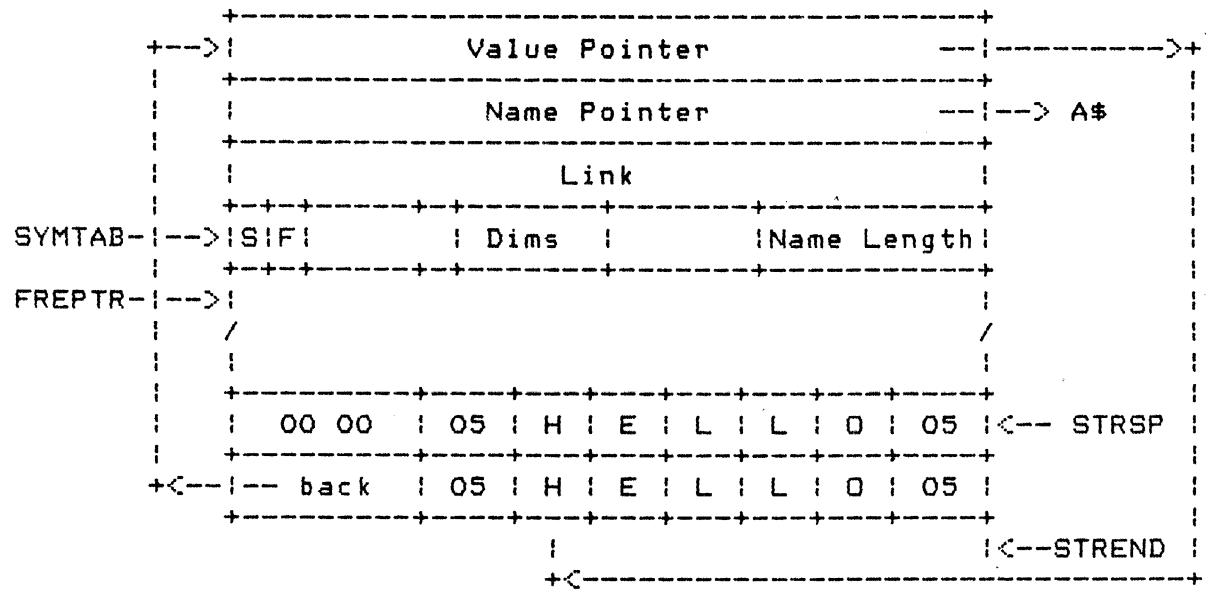


Figure 4.3.3.2.4

The original string is now garbage, as can be seen by the zero backpointer, and the new string has been created and assigned to the variable.

When a garbage collection is performed we begin at the memory location pointed at by STRSP (String space), subtract the length (5) + 3 from that pointer to point at the backpointer for that entry. If it is zero (garbage), the pointer is decremented by one to point at the next string's length and the above subtraction is repeated. If, as in the case above, this string is not garbage (non-zero backpointer), it is moved up in memory so that STRSP and FREPTR point to that last length byte. The value pointer is then fixed up by following it and putting the new address of the string into the value pointer, thus maintaining the loop. This process is repeated until the moveable pointer reaches STREND (string end), which points to the end of the string space. STREND is then reset to point to the new end, completing the garbage collection. In the example, memory ends up looking like:

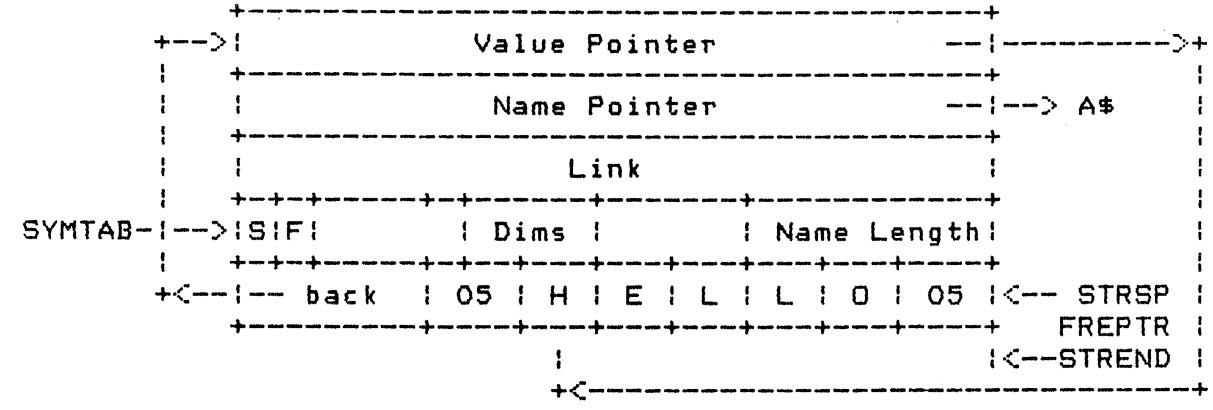


Figure 4.3.3.2.5

Another important facet of the string scheme is the FAC entry which is created by the use of a string. The two types of FAC entries for strings look like:

>001C	>65		Pointer to string	Length of string

Temporary String
Figure 4.3.3.2.6

for a temporary string, and;

Value PtrAddr	>65		Pointer to string	Length of string

Permanent String
Figure 4.3.3.2.7

for a permanent string.

The entry in FAC,FAC+1 is the address of where the pointer to the string came from. In the case of a permanent string it is the address of the value pointer in the symbol table. This, in actuality, is the same value as the string's backpointer. In the case of a temporary string it is the address of SREF (>001C) which is a permanent interpreter variable dedicated to picking up string references. This is critically important item to remember as it is used when a string FAC entry pushed onto or popped off of the value stack.

If a temporary string (FAC/FAC+1=>001C) is pushed onto the stack then the backpointer for the string is changed from >0000 to point at the string pointer in the stack entry making the string semi-permanent. At this point, the string now has an owner and if a garbage collection is performed while the string belongs to the stack, the string won't be lost. When it is

popped off the stack, the pointer in FAC+4,+5 is still pointing at the string, even if it moved during a garbage collection. The backpointer is then cleared to make it a temporary again.

If a permanent string is pushed onto the stack, nothing special is done, as the string's backpointer already points into the symbol table. When the stack entry is popped, however, the string pointer in FAC+4/5 must be updated by using the FAC/+1 entry in case a garbage collection was performed while the entry was on the stack. This is due to the fact that the string had only one owner, namely the symbol table entry, and if a garbage collection was performed while the entry was on the stack the symbol table entry's value pointer would have been updated and not the FAC+4/5 pointer. This problem is taken care of in VPOP to releive all other code from the responsibility of getting the value pointer back again.

4.3.4 Math Package

The Math package used in the 99/4 console supports real-type numeric data with an overall accuracy of at least 13 significant decimal digits. The range for the exponent is E-128 through E+128. This accuracy and range is maintained in the Product 359 BASIC interpreter, as well, since it utilizes the floating-point routines of the 99/4 console.

The format used in this implementation is the following:

Figure 3.12.1

The exponent is contained in the first byte of the floating point value. The most significant bit of this byte indicates the sign of the entire value. If this bit is set, the value of the floating point number is negative and the first two bytes have been negated. If the most significant bit of the first byte is reset this byte contains the actual exponent in excess-64 notation. The exponent given is a base 100 exponent, i.e. to get the actual base 10 exponent, the given exponent has to be multiplied by two. The value of the first byte of the mantissa determines if the actual base 10 exponent is odd or even.

The mantissa is contained in the next seven bytes. Each byte contains two digits in radix 100 notation, i.e. the value of each byte is in the range 0-99. The mantissa is normalized from 01.000000000000 through 99.999999999999. If the value of the first byte of the mantissa is higher than 9 the actual value of the base 10 exponent is odd.

The value, zero, is handled as a special case. Its representation is given by the first two bytes (exponent and first mantissa byte) being zero.

For internal computation the math package uses two extra guard bytes, giving an extra accuracy of four base 10 guard digits. When a computation is completed this ten byte value is rounded to eight bytes.

4.3.5 String Package

The string package of the Product 359 BASIC interpreter consists of several routines dedicated to handling strings. The routines which are dedicated include GETSTR, the system get-string routine, COMPCT, the system garbage collector and LITSTR, the string literal handler of the interpreter. Several other routines have special sections for handling strings. These include: VPUSH and VPOP, the stack push and pop routines; ASSGNV, the variable assignment routine; LTST20, the string comparison routine. The following sections describe some of the above-listed routines as well as how strings are handled within the interpreter. Also, more information may be obtained in section 4.3.3.2.

When a string constant is encountered in a BASIC program it is copied into the string space via the following sequence:

LITSTR - the routine LITSTR (literal string)builds the FAC entry by first putting the string length into FAC+6/+7. Next, GETSTR (get string) is called to allocate a string in the string space and and return the pointer to the string in SREF. The address of SREF is put into FAC/+1 to indicate that the string is a temporary. The pointer to the string is put into FAC+4/+5. The >65 ID for a string is put into FAC+2. Next, the length of the string is checked to see if the string is null. If the length is not zero, the string is copied into the reserved string. If the length is zero, the string does not need to be copied. Note that in copying the string a check must be made to determine if the string is to be copied from a program in the VDP or from a program in the expansion RAM so that it may be copied from the correct memory.

GETSTR - the routine GETSTR (get-string) which must check to see if there enough room in the string space to allocate the string. If there is room it allocates the string. If there is not enough room, it invokes COMPCT to do a garbage collection and then checks again to see if there is enough room. If there is room now, it allocates a string. If not, it issues a * MEMORY FULL error message.

The first thing GETSTR does is pick up the string

length. It then adds four (4) to that length to allow for the backpointer and the two (2) length bytes. It then picks up the end of the string space (STREND) and deducts for the new string. The value stack pointer (VSPTR) is then picked up and has a 64-byte buffer zone added to it to give added protection in ensuring that the string space and the value stack do not collide. A comparison is done to see if there is enough space. If there is room, the exact string length is retrieved (subtract 4). The trailing length byte is now put into the string space at STREND, which points to the first free location. The length of the string is subtracted from STREND to point at the first byte of the string. STREND is now saved in SREF to be the pointer to the string when the allocation of the string is complete. The leading length byte is now put in by decrementing STREND by one to point at it and putting it in. STREND is now decremented by two to point at the backpointer and the backpointer is cleared to indicate that the string is a temporary. STREND is decremented to point to the first free byte again and GETSTR is exited, returning to the caller. The string has now been allocated with the pointer to the string appearing in SREF and the leading and trailing length bytes in place and the backpointer cleared to indicate a null string. The string which the string space has been allocated for has not been copied into the string space. This is the responsibility of the caller after the string has been allocated.

4.3.6 BASIC Statements

This section describes, in detail, how each of the different statements in this BASIC is executed. The statements are grouped together in general categories based upon their general function, e.g. I/O statements are grouped together, control transfer statements are grouped together, etc. As many details as possible are covered, but it is advisable for the reader to look at listings of the code to completely understand how each statement is executed.

4.3.6.1 Input / Output

All of the input / output statements are handled within the FLMGR assembly of BASIC. This section has all I/O statements broken down into either internal I/O, screen or keyboard, and external I/O, device or file.

4.3.6.1.1 Screen / Keyboard

I/O to and from the screen and keyboard via the ACCEPT, INPUT, LINPUT, PRINT and DISPLAY statements of BASIC.

Print Statement

The print-statement can be used to do I/O to either a device or to the display, functioning in a manner similar to the display-statement. Execution of the print-statement begins by checking to see if it is I/O to a device or to the screen. If it is to be output to a device, the PAB is set up and some flags are set to indicate that, later on, when the output is to be actually done, that it is to go to a device. If the output is to the screen, the routine INITKB is called to intilaize the screen for output and to set some flags to indicate that when the output is to actually take place it is to go to the screen.

Control then flows into a global loop which outputs each print-item in the print-list until the end of the line is reached. Inside the loop, a check is made to see if a print-separator preceeds the next print-item in the list. If so, the print-separator is handled in is own particular manner; comma skips to the next field, colon to the next line, and semi-colon to the next column or location. After the print separator is evaluated control flows to the bottom of the global loop. If a print-separator is not present a check is made to see if a reference to the tab function is present. If a reference to the tab function is present, it is evaluated and control flows to the bottom of the global loop. If a reference to the tab function is not present, control flows into the code which parses the expression and outputs it.

After the expression is parsed, a check is made to see if output is being done to an internal-type file (does not include output to the screen), and if so, numeric values are placed into eight-byte strings and explicit strings are left as they are. Finally, a check is made to see if a string or numeric is being output. If a string is being output, it is output using OSTRNG. If a numeric is being output, it is converted to a string and output, also using OSTRNG. OSTRNG is a routine which outputs any pending records to a device and then takes care of outputting the current item. It 'chunks' up any items which are too long for the device or screen, repeatedly putting out as many records as necessary to get the entire item out.

After the print-item has been output, a check is made to see if a print-separator or an end-of-statement occurs next, and, if neither is present, an error occurs. If a print-separator is present it is handled and the entire loop is repeated until the end-of-statement is reached. When the end-of-statement is reached, an XML CONT instruction is executed to return to the statement dispatcher to process the next statement or to return to top-level.

In pseudo-code, the entire execution of the print and display statements can be described as follows, with the label, PRINT, being the entry-point for the print-statement and the label, DISPLAY, being the entry point for the display-statement.

```
DISPLAY INITIALIZE_SCREEN
        EVALUATE_OPTIONS
        GOTO PRINT1
PRINT    IF DEVICE-OUTPUT THEN
           INITIALIZE_PAB
         ELSE IF SCREEN
           INITIALIZE_SCREEN
         END IF
PRINT1   REPEAT
           IF SEPARATOR THEN
             HANDLE_SEPARATOR
           ELSE
             IF TAB THEN
               HANDLE_TAB
             ELSE
               PARSE_EXPRESSION
               IF INTERNAL-FILE-TYPE THEN
                 IF NUMERIC-ITEM THEN
                   CONVERT_TO_8-BYTE_STRING
                 END IF
               END IF
               IF STRING-ITEM THEN
                 OSTRNG
               ELSE IF NUMERIC-ITEM
                 CONVERT_TO_STRING
                 OSTRNG
               END IF
               IF NOT SEPARATOR THEN ERROR
               HANDLE_SEPARATOR
         UNTIL END-OF-LINE
         CONT
```

Display Statement

The display-statement is executed in exactly the same manner as a print-statement to the screen except that additional options are available to allow for erasing the screen, random access to the screen, the issuance of a "beep" and limitation of the output field size.

Execution of the display-statement begins by initializing I/O for output to the screen. After I/O has been initialized for the screen, the possible options are checked for, evaluated and finally the information to be displayed is placed on the screen. Note that each option may appear in a single display-statement only one time. The code which handles each clause first checks

to be sure that the clause has not already been used once and if it has not, it is evaluated, otherwise, an error occurs. Each clause handler also sets a flag after the clause has been evaluated to indicate that one occurrence of the clause has already appeared. The erase-all option is the first one evaluated. If the keywords, ERASE and ALL, appear in the display-statement, the screen is cleared and a flag is set to indicate that the erase-all option has been used and control returns to the top of the option evaluation routine and checks for another option. After checking for an erase-all clause, the beep-clause is checked for and if it is present, a flag is set to indicate that a beep-clause has been used and control again returns to the top of the option evaluation routine. After checking for a beep-clause, the at-clause is checked for. If it is present, the row and column numbers are evaluated, placed in the ranges of 1 to 24 and 1 to 28 respectively and the correct screen address is calculated. Again, a flag is set to indicate that an at-clause has been used and control returns to the beginning of the option evaluation routine. After checking for the at-clause, the size-clause is checked for and if it is present, is evaluated and saved for later use.

The flags which are set when the options to the display statement are evaluated are set in the CPU memory location PABPTR. When screen I/O is specified PABPTR is not used to point to a PAB since none is used but is used to indicate the options selected. Six bits in PABPTR have been assigned meanings as shown below.

BIT	MEANING
7	Don't blank current field (negative size)
6	Validate used (accept-statement)
5	(unused)
4	(unused)
3	Size-clause used
2	At-clause used
1	Beep used
0	Erase-all used

Once all of the possible options for the display-statement have been evaluated, control flows into the code which handles the print-statement, as the display-statement and a print-statement to the screen function in exactly the same manner. A complete description of how the print-statement is executed may be found in the preceeding section.

Using Clause

The using-clause is an option which is available for both the display and print statements. Evaluation of the using-clause occurs by first picking up the format information and then using that information in outputting the specified values.

The format information may be referenced in either of two ways. First, an image-statement may be used and the line number of the image-statement may be referenced in the using-clause. In this case, the routines which are normally used in the searching and evaluation of read and data-statements are used to search for and read the format specification. If an image statement is not used, the format information can be supplied by the use of a string expression following the using keyword. In this case, a standard parse is done to get the format string for use in specifying the format of the output.

After the format string has been picked up, a "working" copy of it is created. This "working" copy will eventually become the output string after the necessary string and/or numeric values have been parsed. This copy is used to count the number of number-signs supplied to indicate the field to be used for the output information.

In the case of numerics, this includes both the digits to the left of the decimal point and the digits to the right of the decimal point. Also, in the case of numerics, the number of places needed for the displaying of the exponent (circumflexes) are counted and the necessity of including an explicit sign is checked for. The numeric value to be output is then parsed and it is converted into a string by using the information gathered in examining the format string. The converted number is then copied into the "working" copy of the format string.

In the case of outputting strings with a particuluar format, the string to be output is copied into the "working" copy of the format string, left-justified, with spaces filling to the right of the string if the format string is longer than the output string. If the string to be output is too long for the format specified, the "working" copy string is filled with asterisks to indicate that the field is not large enough for the string to be output properly.

After the entire "working" copy of the format string has been converted into the output string, it is output to either the screen or to the device specified.

Input Statement

The input-statement is separated into two separate parts. One part provides for input from the keyboard and the other provides for input from a device. This section describes input from the screen.

After it has been determined that input in coming from the keyboard (no file-clause or file number 0) a check is made to insure that there is enough room on the screen for the question mark to be placed on the screen and if there is not the screen is scrolled to create room. The question mark prompt is then

displayed.

Next the input statement is scanned to pick up all of the variables which are to receive values and to push the special entries for each (created by a call to SYM) onto the stack. After all of the variable entries are on the stack, the rest of the line on the screen from which the input is to come is cleared and the prompting tone is sounded. The routine, READLN, is called to read the input line from the screen and to allow all of the editing features for the input line. After the line has been read, it is crunched by calling the routine SCDATA, which is a subroutine used by CRUNCH to crunch data-statements. Crunching an input line is essentially the same as crunching a data-statement. After the line has been crunched, the screen is scrolled and a check is made to see if the number of arguments input matches the number variables in the input-statement. If the number of arguments does not match the number of variables, a warning message is issued and the input is tried again.

At this point an assignment loop is entered which rescans the input line, in case a subscript is being input, and the values read from the screen are assigned to the variables in the input list until the end of the input-statement line is reached. When all of the values have been assigned, an XML CONT instruction is executed to return to the parser. The following diagram describes, in general terms the structure of the input statement.

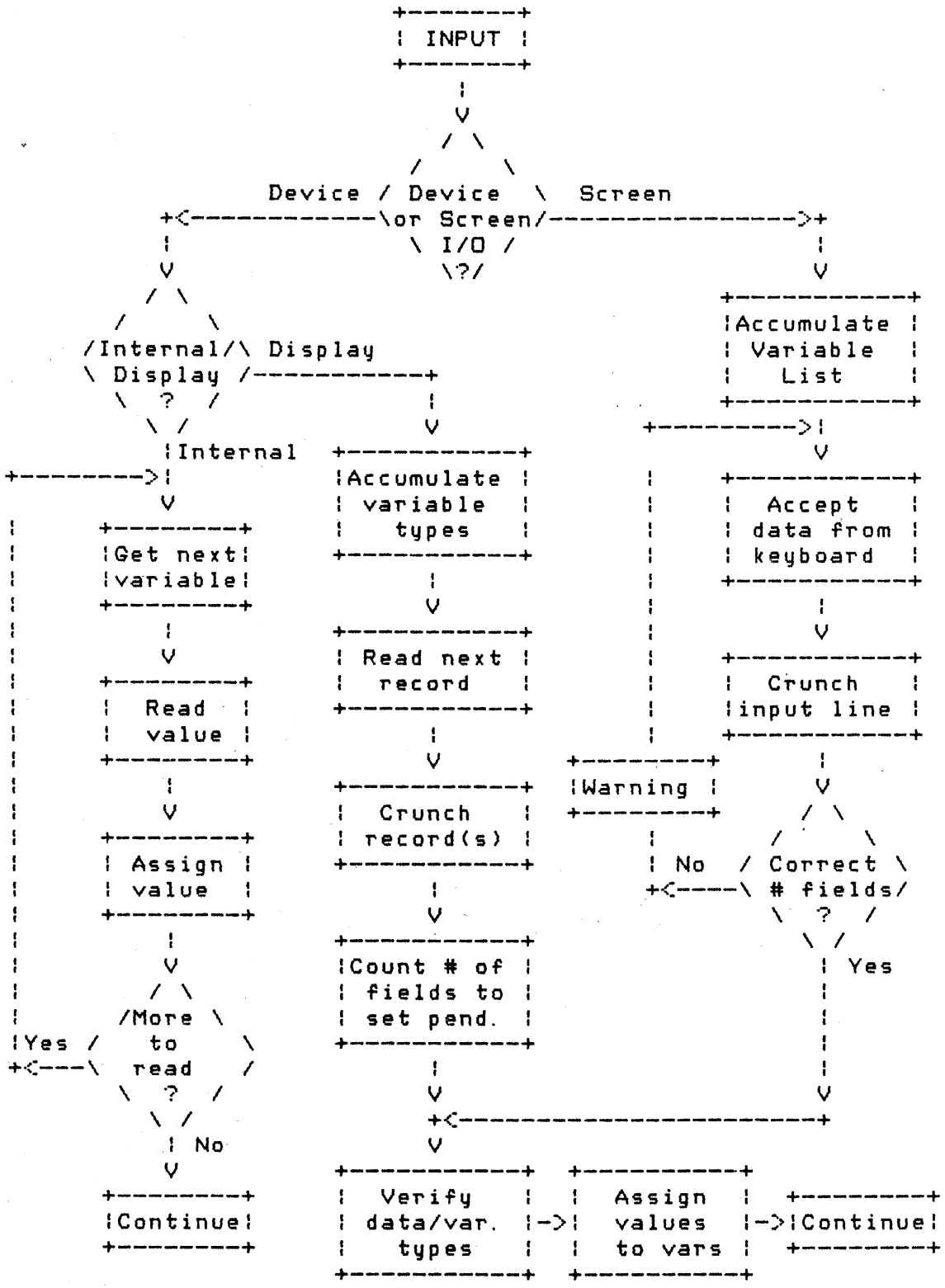


Figure 4.3.6.1.1.1

Linput Statement

The linput statement is used to take input data from either the keyboard or an external file without any interpretation of the data at all and assign it to a string variable. Execution of the statement involves first calling a subroutine to take care of initializing for input from the keyboard. If a file number is present then input is initialized from the particular file. In the case of file I/O the file type of the device must be display or an error occurs.

The information about the variable specified in the statement is accumulated and the variable is insured to be a string variable.

If the linput is directed to come from a device and a new record must be read, then it is read, put into a string and assigned to the variable. If a partial record is available (left by execution of a previous INPUT statement) then the screen offset is removed from each character in the record, the value is copied into a string in the string space and assigned to the variable.

If the linput is directed to come from the keyboard, then any input prompt is handled and the READLN routine is called to accept the input. Then the screen offset and all edge-characters are removed from the input line and the resulting string is assigned to the variable.

Accept Statement

The accept-statement is used for inputting data from the screen(keyboard) at a random location. It also provides for the validation of the input data, the issueing of a "beep" from the tone generator, restriction on the size of the input field allowed and the erasure of the entire screen.

Execution of the accept-statement is done by first calling the same routine used by the display-statement to evaluate the common options, at, beep, size and erase-all, and then checking to see if a validate-clause is present. If a validate-clause is present, it is evaluated by first looking for the verification keywords, UALPHA, NUMERIC or DIGIT and if any or all of the keywords are present, special flag bits are set to indicate which of the verification keywords have been specified. Also, if any strings are supplied which contain characters that may also be accepted, they are parsed and the string entries which are returned by the parser in the FAC are pushed onto the stack and saved for validation later on. The process of looking for the keywords and/or strings is repeated until the end of the validate list is reached (a right-parenthesis is encountered). Thus, the options may appear in any order within a statement.

The READLN routine is then called with the appropriate flags set to indicate it must do validation of the input data. READLN then checks each character entered against the set of legal characters and if in illegal character is entered, it is rejected. After the input line has been read (validated also), the value is assigned to the variable specified. The screen is then scrolled, if necessary, and control returns to the parser.

4.3.6.1.2 Device / File

An extensive device/file I/O system is supported by the Product 359 BASIC interpreter. The file management interface is discussed in detail in the Home Computer File Management Specification and the reader is referred to that document. Discussed here, briefly, are the Peripheral Access Blocks (PABs) and how the OPEN, CLOSE, OLD, and SAVE statements are executed.

Peripheral Access Block Definition

All DSRs are accessed through a Peripheral Access Block (PAB). The definition for these PABs is the same for every peripheral thus providing a device-independent I/O interface. The only difference between peripherals, as seen by BASIC, is that some peripherals will not support every option provided for in the PAB.

All PABs are physically located in VDP RAM. They are created before the OPEN call, and are not to be released until the I/O has been closed for that device or file.

Figure 4.3.6.3.1 shows the layout of a standard PAB with the additions to it by the interpreter to manage the PABs in memory (first eight bytes). The PAB has a variable length, depending upon the length of the file descriptor.

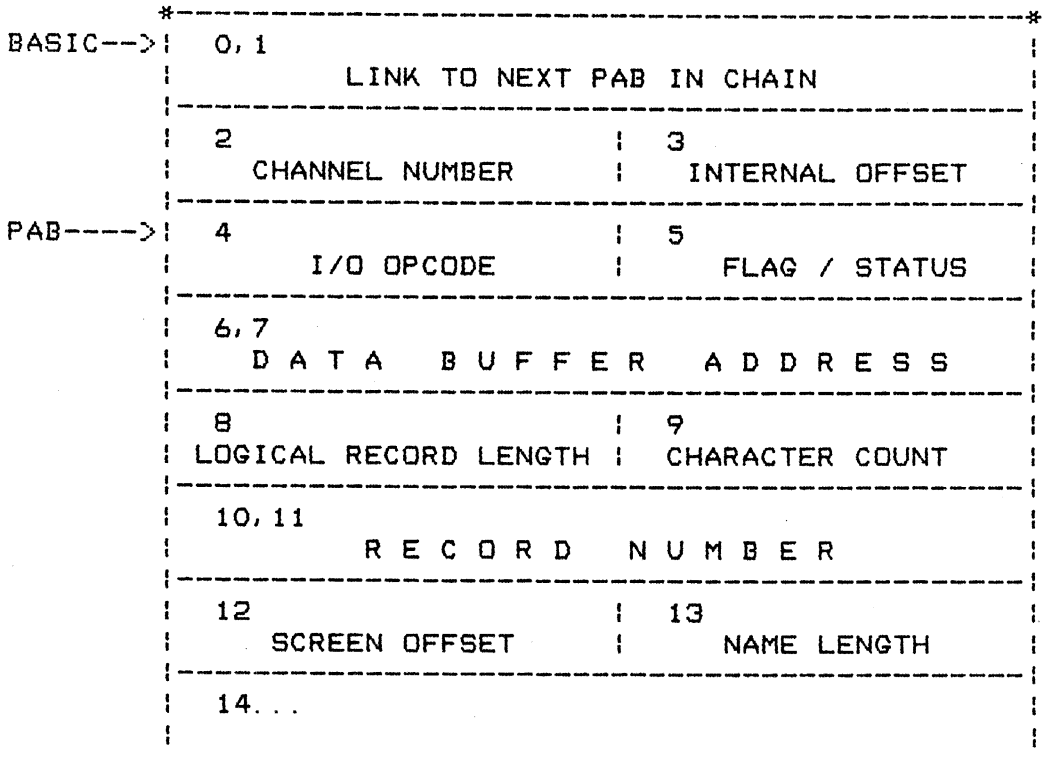


Figure 4.3.6.1.3.1 BASIC PAB Layout

The meaning of the bytes within the PAB are described below.

Byte		Meaning
0	-	I/O opcode - Contains opcode for the current I/O-call. A description of the valid opcodes will be given later.
1	-	Flagbyte/status - All the information the system needs about file-type, mode of operation, and data-type, is stored in this byte.
2,3	-	Data buffer address - Address of the data buffer the data has to be written to or read from.

4 - Logical record length - Indicates the logical record length for fixed length records, or the maximum length for a variable length record (see flagbyte).

5 - Character count - Number of characters to be transferred for a WRITE opcode, or the number of bytes actually read for a READ opcode (not equivalent to INPUT and OUTPUT mode).

6,7 - Record number - Only required if the file opened is of the relative record type. Indicates the record number the current I/O operation is to be performed upon (this limits the range of record-numbers to 0 - 32767). The highest bit will be ignored by the DSR.

8 - Screen offset - Offset of the screen characters in respect to their normal ASCII value.

9 - Name length - Length of the file descriptor following the PAB.

10+ - File descriptor - Devicename and, if required, the filename and options. The length of this descriptor is given in 9.

BASIC PAB Additions

Aside from the control information contained within the PAB, BASIC adds four (4) more bytes to the top of the PAB for specific BASIC related control information.

The additional four bytes contain control information BASIC needs for its internal PAB linkage structure. The PABs within the BASIC control structure, form a linked list. The last PAB in the list has a zero ("0") link.

The first 2 bytes in the BASIC PABs contain the link to the next PAB. The next byte contains the actual BASIC channel or file number (1-255). The next byte contains the offset of the current data-pointer within the data-block.

The offset indicated in the third byte of the BASIC PAB indicates the position of the current data pointer within the data buffer given in bytes six and seven of the PAB. If byte

three equals zero, the current data buffer is "blank", i.e. if in "read" mode, a new buffer has to be read in before any further processing; in "write" mode, the entire buffer is still available for data storage.

If byte 3 is non-zero, it contains an "offset" within the data buffer. Added to the start address of the data buffer, it will give the actual address of the first data-byte to be read or written. This is only the case if there are pending print operations or the most recent INPUT ended on a comma. In all other cases, byte 3 will be zero.

I/O Operations

The valid opcodes which may appear in a PAB are shown in figure 4.3.6.1.3.2. The following sections describe the general actions invoked by an I/O-call when the open, close, load and save I/O-opcodes are used.

Opcode	Meaning
00	OPEN
01	CLOSE
02	READ
03	WRITE
04	RESTORE/REWIND
05	LOAD
06	SAVE
07	DELETE
08	SCRATCH RECORD
09	STATUS

Figure 4.3.6.1.3.2 I/O Opcodes

OPEN

The OPEN operation should be performed before any data transfer operation. The file remains open until a CLOSE operation is performed. The mode of operation for which the file has to be opened should be indicated in the flagbyte of the PAB. In case this mode is UPDATE, APPEND or INPUT, the record length will be returned in byte 4. Any given non-zero record length will be checked against this stored length. For OUTPUT the record length can be specified, or a default can be used by specifying record length zero.

Execution of the OPEN-statement begins by picking up the BASIC channel number or Logical Unit Number (LUNO). Once it does

this a temporary PAB is built in the CPU RAM until the device can be verfied open, at which time the PAB is copied into the VDP RAM and made permanent. When the framework for the PAB has been done, the options are parsed and put into the PAB to customize it for the particular device being opened. At this point, the Device Service Routine (DSR) is called to actually open the device or file. If the DSR does not encounter an error the PAB is made a permanent structure and control returns to the parser via the execution of an XML CONT instruction.

CLOSE

The CLOSE operation informs the DSR that the current I/O sequence to that DSR has been completed.

Execution of the close operation is a relatively simple operation. The first thing done is the BASIC channel number is picked up and the correct PAB is found in the PAB list. If the device was opened for output, any pending output that may exist is output. The close op-code or the delete op-code (if delete is specified) is placed in the PAB and the DSR is called. After the CLOSE operation has taken place, the PAB is deleted by a routine called DELPAB. This removes the PAB from the PAB list and releases the memory so that it may be allocated for another PAB, a symbol table entry or the string space.

If a file or device is opened for OUTPUT or APPEND mode, and EOF (End Of File) record is written to the device or file, before disconnecting the PAB.

OLD

The OLD operation loads an entire memory image from an external device. All of the control information BASIC needs is concatenated to the program image before calling the DSR to save the program so that when the program is loaded with the OLD command all necessary information is available to the interpreter.

The OLD operation is a stand alone operation, i.e. the OLD operation can be used without a previous OPEN operation.

For the OLD operation, the PAB contains the following information :

Bytes 2,3 : Start address of the memory dump area

Bytes 6,7 : Number of bytes available.

Aside from the I/O opcode and the file descriptor, no other

PAB-entry is required for the OLD operation.

Execution of the OLD command involves three phases. First, the PAB is built. Then the DSR is called to actually read in the program from the device specified by the name. After the program has been read into the VDP RAM the starting and ending line number table pointers are retrieved from the memory image and are used to locate the program in memory and fix all of the line pointers within the line number table. After this is all completed control returns to top-level to await input from the user.

See the Product 359 BASIC Interpreter Expansion RAM Support Software Specification for information on how a program is loaded into the expansion memory.

SAVE

SAVE is the complimentary operation for OLD. It is used for writing memory images to a device or file. All necessary control information is linked to the memory image, so that the information plus program image use one contiguous memory area. As with the old operation, only a small part of the PAB is used. Aside from the usual information (I/O opcode and filedescriptor), the PAB contains :

Bytes 2,3 : Start address of the memory areaa.

Bytes 6,7 : Number of bytes to be saved.

Execution of the SAVE command involves first checking to see if the program is to be saved in the MERGE format. If the MERGE option is specified then the file is opened and each line of the program is written to the file, individually, with the line number, length and text all appearing in a single record. The first record written includes control information to indicate that this file is actually a program saved in the MERGE format. After all of the program lines have been written to the file, the file is closed and control returns to top level.

If a program is to be saved in the normal program format the correct PAB is built and the starting (STLN) and ending (ENLN) line number table pointers are appended to the program image in the VDP RAM so that they can be saved also. A "checksum" is created by XORing the STLN and ENLN pointers together is also created to be saved with the program image. A check is made to see if the program is to be saved in the protected format and, if so, this "checksum" is complemented so that when an attempt is made to load the program from the device the fact that it is protected can be detected. Note that by doing this a program saved in the protected format cannot be loaded by the 99/4 BASIC because the "checksum" will not be

correct and an error will occur. When all this has been completed, the DSR is called to actually write the memory image to the device. Control then flows back to top-level to await a command from the user.

See the Product 359 BASIC Interpreter Expansion RAM Support Software Specification for information on how a program is loaded into the expansion memory.

MERGE

The MERGE command is used to load a program saved in the merge format back into the computer's memory as if the program were being entered from the keyboard. Execution of the MERGE command involves first opening the file and checking to see if it is a program saved in the merge format. If it is not, the file is closed and an error occurs. If the file is a program saved in the merge format, then each line (record in the file) is read and then edited into the program by calling the EDITLN routine, sequentially, until the entire file has been read. When the entire program has been edited into memory, the file is closed and control returns to top level.

4.3.6.1.3 READ / DATA and RESTORE

READ and DATA statements allow a BASIC program writer to store data in a program and retrieve it easily, using a minimum of space. In order to properly handle READ and DATA commands the interpreter, when a program is being scanned, the static scanner looks for the first occurrence of a DATA statement and, if one is found, the address of the first item of that DATA statement is stored in the global variable DATA. The address within the line number table of the line in which the DATA statement has been found is stored in the global variable LNBUF. In case no DATA statement is present in the current program segment, the high order byte of DATA is set to zero (0). Thus, when program execution actually begins the data pointers are already set up properly and nothing additional needs to be done before execution can begin.

Each READ statement reads one or more items from a DATA statement. Items are separated by commas. If, during a READ, an end-of-statement is encountered, the variable LNBUF is used to scan subsequent program lines searching for another DATA statement. If another DATA statement is encountered, the DATA pointer is set up for that particular DATA statement and the READ is completed.

After all DATA statements in the program have been used, the high byte of DATA is set to 0, indicating that no more data

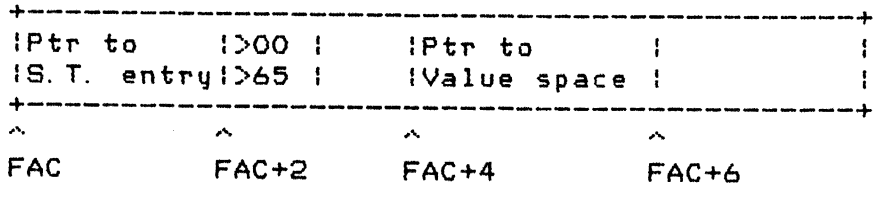
is available. Execution of a READ statement after this will result in an error.

The DATA pointer can be reset to any DATA statement within the program segment by using the RESTORE command. In the RESTORE command an optional line number can also be specified. This line number indicates the line at which the DATA scan is to start. If the line number specified is not contained in the program, the RESTORE defaults to the first line in the program which contains a DATA statement.

4.3.6.2 Assignment

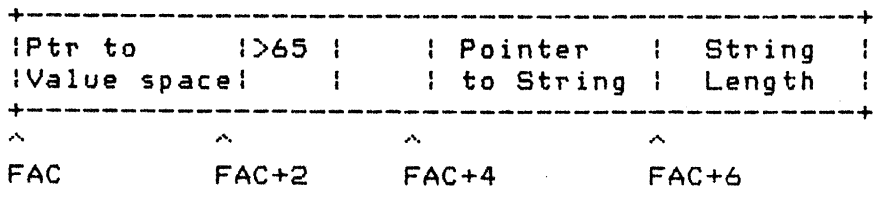
There are two separate types of assignment statements, numeric assignments and string assignments. Assignment of values to variables is handled by the ASSGNV (assign value) routine which is located in the BASSUP assembly of BASIC. In preparation for the use of ASSGNV, SYM (symbol name) and SMB (symbol value) must also be called to prepare the variable for assignment. SYM serves two functions. First, it picks up the variable name from the statement text, advancing the text pointer, PGMPTR, past the name and searches the symbol table for the variable. Second, it places the address of the symbol table entry into the FAC. If at any time during the name accumulation or symbol table search an error occurs, the operation is aborted and an error message is generated. SMB is then used to find where the value being assigned to the variable must be placed.

In the case of numeric variables SMB must comprehend the fact that the values for numeric variables may not reside within the symbol table but may be contained in the expansion RAM peripheral. It does this checking the first byte of RAMTOP to see if the expansion RAM is present. It then begins with either the address of the value space within the symbol table or with the address of the value space in the expansion RAM. If an element of an array is being referenced it then calculates the correct offset into the value space by evaluating the subscript(s). SMB places the address of where the value is to go into FAC+4. An identification byte of >00 is added in FAC+2 to indicate to ASSGNV that the symbol is a numeric. The 8-byte FAC entry, after SYM and SMB have been called for a numeric variable looks like:



Numeric variable entry after SMB
Figure 4.3.6.2.1

In the case of a string variable SMB must construct a somewhat different FAC entry due to the fact that a pointer is needed to the string value. The entry is built by putting the pointer to the value space into FAC and FAC+1 and putting the pointer to the string in FAC+4 and the length of the string into FAC+6. Finally, the ID byte of >65 is added to the entry to identify it as a string variable entry resulting in an entry for a string varaible that looks like:



String variable entry after SMB
Figure 4.3.6.2.1

After the FAC entry has been constructed it must be pushed onto the value stack and the value to be assigned must be placed into the FAC. The following sections describe what unique things must be done in order to assign the value to the variable depending upon whether it is a numeric assignment or string assignment.

4.3.6.2.1 Numerics

ASSGNV first pops the top entry off of the stack into the ARG area of the FAC. This entry should be the entry constructed by SYM and SMB. If ASSGNV determines that a numeric argument is being assigned to a numeric variable it copies the eight bytes of the FAC into the symbol table at the location specified by the pointer picked up from ARG+4.

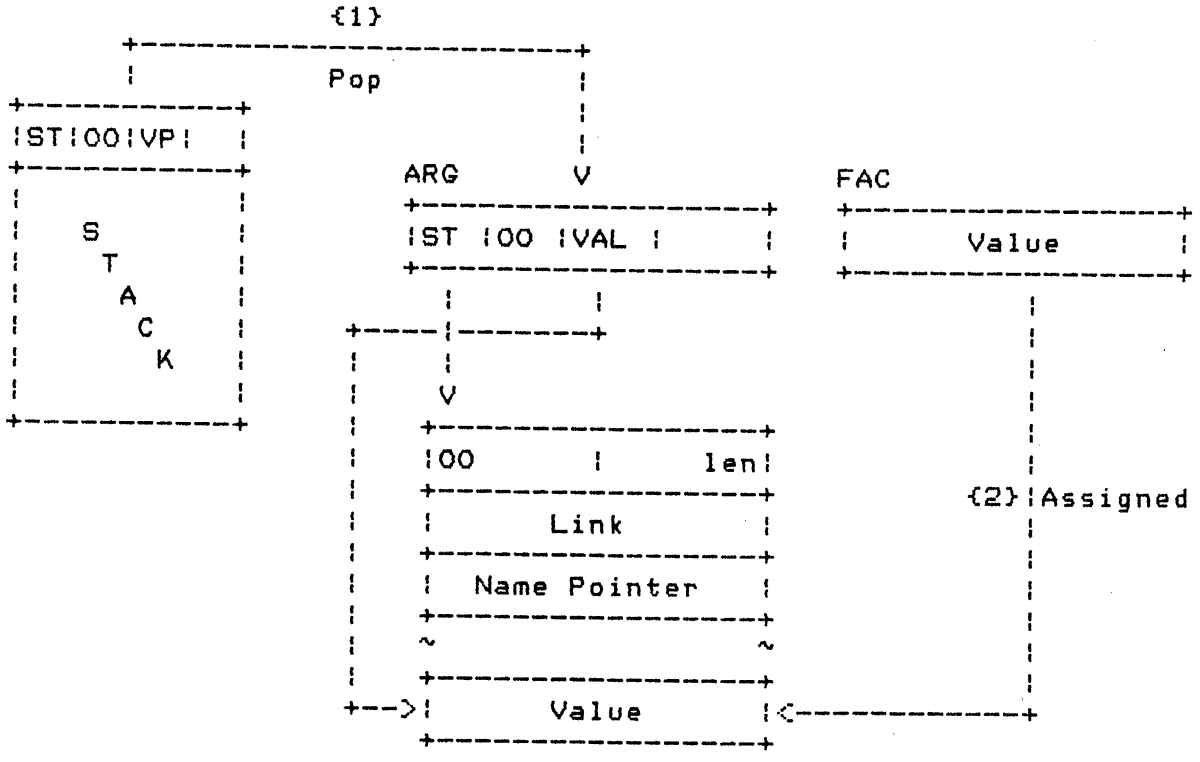


Figure 4.3.6.2.1.1

After the value has been copied into the correct place in the symbol table, ASSGNV returns to its caller.

4.3.6.2.2 Strings

ASSGNV first pops the top entry off of the stack into the ARG area of the FAC. This entry should be the entry constructed by SYM and SMB. If it is not, an error occurs. If ASSGNV determines that a string argument is being assigned to a string variable it must do several things to correctly make the new assignment. First, it checks to see if the variable currently has a value assigned to it. If it does it must free the string assigned to it so that it can be garbage collected at a later time. Then, if the string to be assigned is currently assigned to a symbol (the FAC entry indicates that the string is not a temporary), a new string is created that exactly matches the string to be assigned and this new copy is assigned to the symbol pointed to by the pointer in ARG.

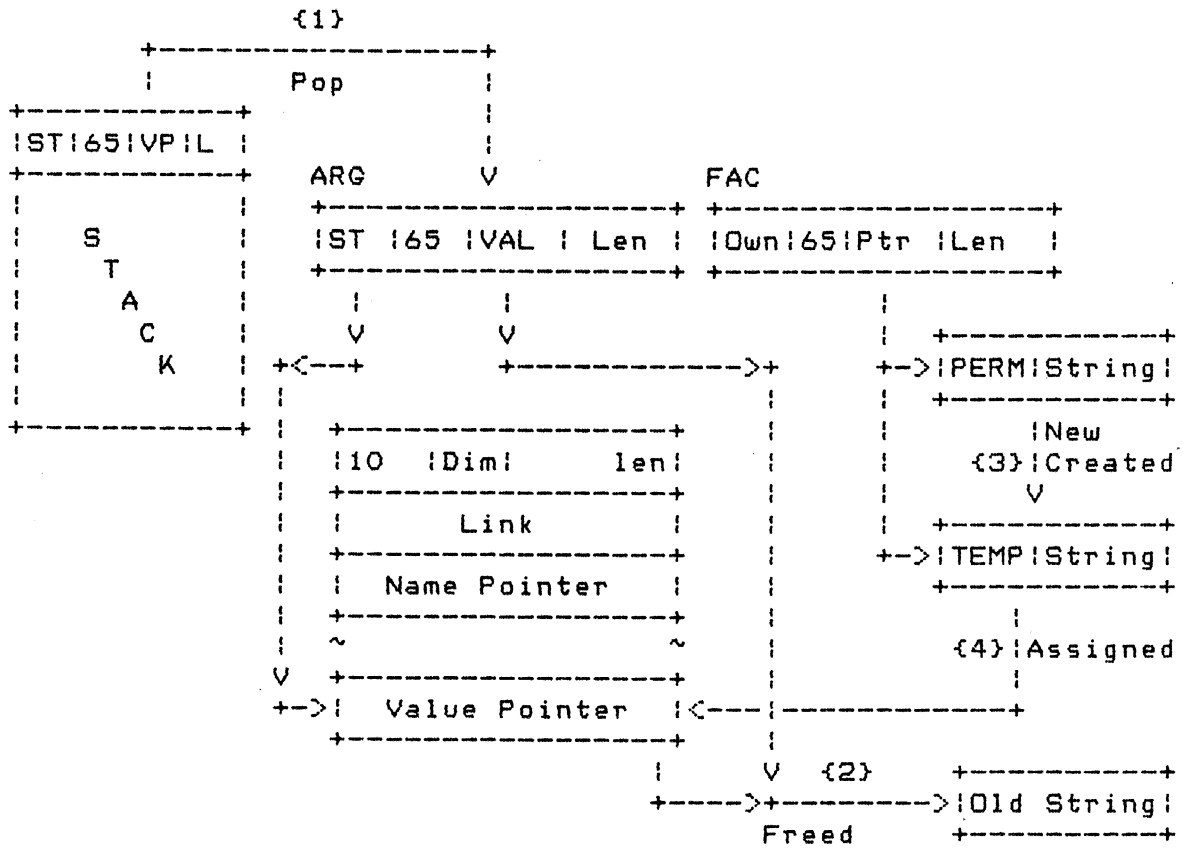


Figure 4.3.6.2.2.1

After the new string has been assigned to the symbol table entry, ASSGNV returns to its caller. Additional information on strings appears in Section 4.3.3.2.

4.3.6.2.3 LET

Assignments to variables may take place in many ways but the let-statement is the primary means. Execution of the let-statement is really quite simple. Initially, a counter, which will indicate how many variables are to be assigned the value to the right of the equal sign. Next, the symbol routine, SYM, is called to get the pointer to the name of the symbol which is to receive the new value and to search the symbol table for its entry. SMB is then called to get the pointer to the correct place in the variable's value space. The entry created by SYM and SMB in the FAC is then pushed on the stack and the counter of the number of assignments to be made is incremented. A check is made to see if a comma token is present, which would indicate that another variable is to be assigned a value, and, if it is, it is skipped and the SYM, SMB process is repeated until an equal sign is encountered. When the equal sign is

encountered, the value to be assigned to the variable(s) is parsed.

Another loop is then executed which repeatedly assigns the value parsed to the variables by using the ASSGNV routine described in the preceeding two sections and decrementing the counter set up when the left-hand side of the let-statement was scanned until all of the variables have been assigned the new value. There is one minor difficulty arising when multiple assignments to string variables are being made because multiple copies of the string parsed must be created. This condition is taken care of by making the FAC entry for the string being assigned look like a symbol-to-symbol assignment of strings is taking place to the ASSGNV routine so that a new copy of the string is created. This is achieved by changing the two pointers in the FAC which indicate where the string comes from to make the string that is assigned to the next variable in the list be the one which was most recently assigned. Figure 4.3.6.2.3.1 describes this situation in visual terms.

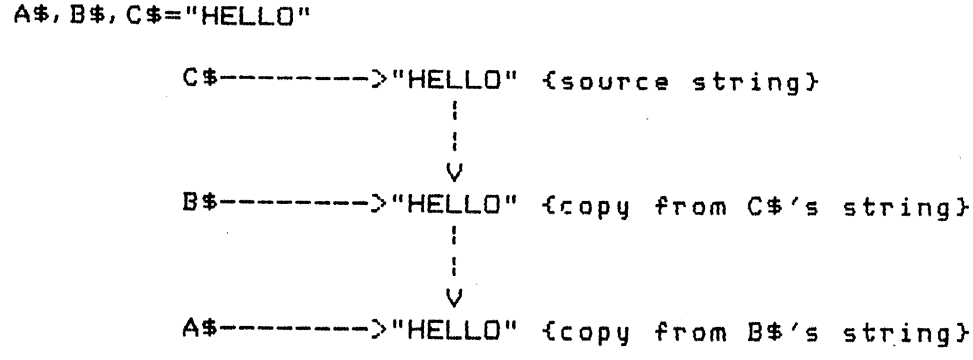


Figure 4.3.6.2.3.1

In this case the source string is assigned to the variable C$. The FAC entry is modified so that it now indicates that the string to be assigned to B$ is the string which belongs to C$. ASSGNV is called again and it creates a new string because the source string indicated by the FAC entry says that a symbol-to-symbol assignment of strings is about to take place. When ASSGNV returns, the process is repeated by changing the FAC entry so that it indicates that the string to be assigned to A$ is the string which belongs to B$, and so on. This repeats until all of the variables specified have been assigned the new value.

4.3.6.3 Control Transfer

The BASIC interpreter normally executes the statements of a program in a sequential manner. When it is finished executing one statement, control flows to the next statement in the program unless one of the statements described in this section dictates otherwise. The statements described here can cause

unconditional branching, conditional branching, branching with return, and looping to occur.

4.3.6.3.1 GOTO

The execution of a goto-statement consists of two parts. First, is the identification of the goto-statement and second is the actual branching to the statement specified by the line number provided.

Identifying a goto-statement is somewhat complicated by the fact that it can have two distinct forms, GOTO and GO TO. When the GOTO form is used the address of the statement handler is picked up directly from the statement table. In the case of the GO TO form, a check must be made after the GO has been looked up in the statement table to see if the token following it is a TO or a SUB. If a TO is found after the GO then control flows to the same location as in the case of the GOTO.

Finding the line to resume execution at is handled by a routine which is utilized by the goto-statement, the on-goto-statement, the gosub-statement, the on-gosub-statement and the then and else-clauses of the if-then-else-statement. Notice that the goto-statement is a specialized case of the on-goto-statement without the index. It should be obvious to see that if the index for a goto-statement can be assumed to be zero, then the goto-statement can be treated exactly as an on-goto-statement after the index has been evaluated. This is exactly what is done by the interpreter. A dummy index (R3) is set up and from here the statement is executed in exactly the same manner as the on-goto-statement described in section 4.3.6.3.3. Refer to that section for how the goto-statement is completed.

4.3.6.3.2 GOSUB and RETURN

The gosub-statement is handled in a very similar manner to the goto-statement. Initially, the same procedure is used to get to the gosub statement handler as is used by the goto-statement (see preceeding section). After control has reached the gosub-statement handler, a special stack/FAC entry is built which saves the current line number table pointer (EXTRAM) so that execution may resume at the statement following the gosub-statement when a return-statement is executed. A dummy on-index is created just as with the goto-statement and control flows into the common code used by the goto, on-goto and on-gosub statements. See the preceeding and following sections for how execution of the gosub-statement is completed and for more information.

Execution of the return-statement involves popping an entry off of the value stack, looking for a gosub entry. If a gosub entry is not found, succeeding entries are popped off until one is found or until VPOP attempts a stack-underflow which causes an error. If a gosub entry is found, EXTRAM and PGMPTR are restored from the entry and execution resumes there by branching to NUDEND to get the next statement.

4.3.6.3.3 ON GOSUB and ON GOTO

The on-goto and on-gosub-statements are handled in almost the exact same way as the goto and gosub-statements, respectively. The on-goto and on-gosub statements are the generalized cases of the goto and gosub statements alluded to in the previous two sections.

When the on-statement token is encountered control flows to a section of code which first checks to see what type of an on-statement is being executed. If an on-error, on-warning, or on-break statement is being executed it is handled by the appropriate code as described in sections 4.3.8.4, 4.3.8.5 and 5.1.3, respectively.

If an on-goto or on-gosub statement is being executed, control flows to the code which evaluates the index, converts it to an integer and then proceeds into the code which picks out the goto/gosub portion of the statement. The index for the statement is maintained in a register (R3) and when the code which looks for a particular line in the program is entered it uses the index to select the correct line number of the line-number -list by using the following method. It first checks for the special line-number token. If one is not present an error occurs. If one is present, the index is decremented to see if the correct line number has been found. If the count is greater than zero, the correct line number has not been encountered and the process is repeated until the index reaches zero, indicating that the correct line has been found. Once the correct line has been found then the common code for finding the line in the line number table is entered just as in the case of the goto and gosub-statements without the indexing.

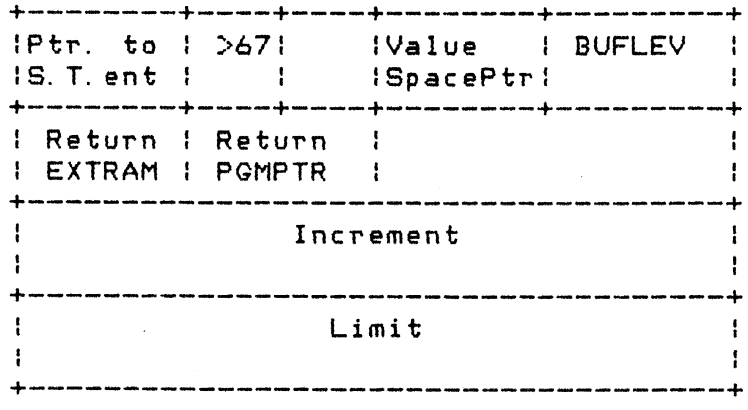
4.3.6.3.4 FOR and NEXT

The for-to-step statement and the next statement are defined in conjunction with each other. The physical sequence of statements beginning with a for-to-step statement and ending with a next statement is termed a for-block. For-blocks can be physically nested, (i.e., one can contain another), but they cannot be interleaved (i.e., if a for-block contains any portion of another for-block, it must contain all of the second block).

Physically nested for-blocks cannot use the same control variable.

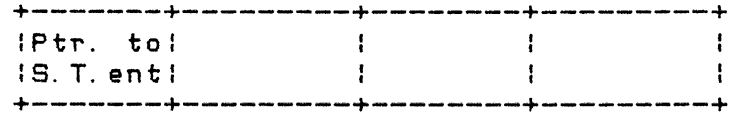
FOR Statement

When a for-statement is encountered, the statement is processed, left-to-right, and a stack block is generated, which is left on the stack to picked up by the next-statement. The stack-block has the form of:

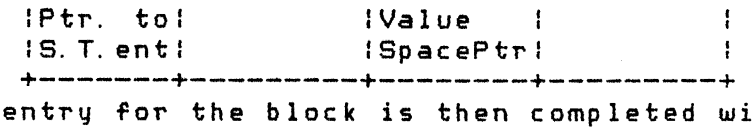


The code which generates the stack block is fairly complex in its manipulations of the stack and is complicated by the fact that in order to maintain ANSI compatibility, the initial value must be maintained while the limit and optional step values are parsed. A step-by-step series of diagrams follows the development of the stack-block below. It is recommended that the reader have a copy of the Assembler listing when following the development below.

The for-code is entered with R8 containing the first character of the index symbol's name. The routine SYM is called to get the pointer to the symbol's symbol table entry. The FAC entry returned by SYM now looks like:



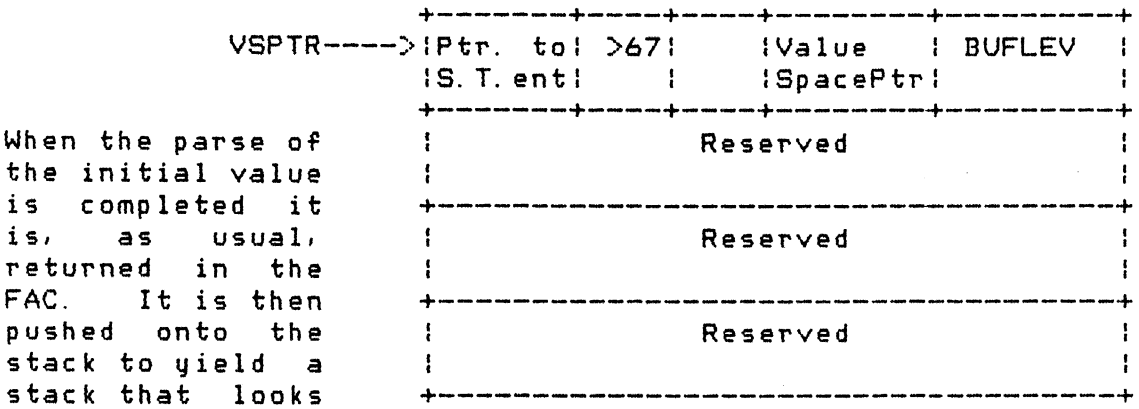
Next, SMB is called to get the pointer to the symbol's value space. The FAC entry now has the form of



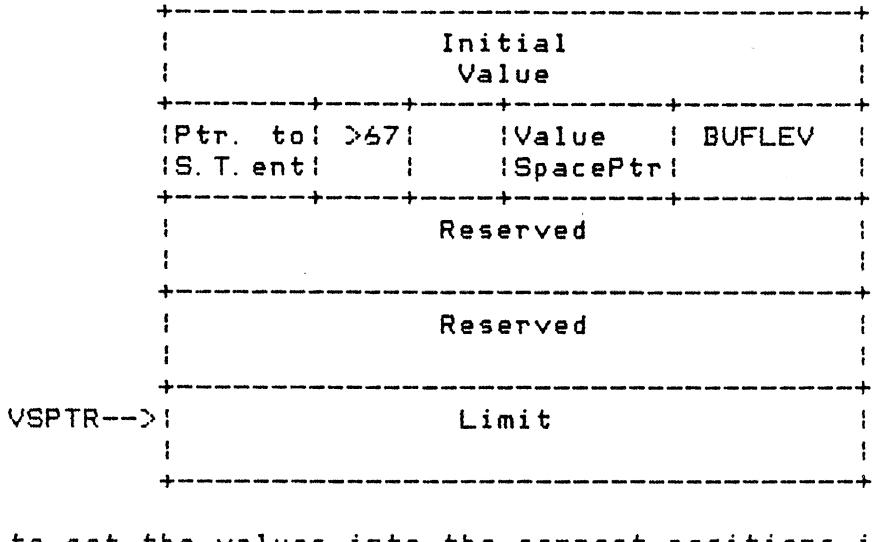
The header entry for the block is then completed with the stack for-block identification, >67, and BUFLEV is added to allow checking for crunch-buffer integrity.

Next, a search of the existing stack entries is made, looking for a for-block with the same index variable. If one is found, it is deleted; if not, the stack remains unchanged.

Next, the actual use of the stack begins. Twenty-four (24) bytes are reserved for the limit, increment and return information, and the identification entry is pushed via a BL @PSHPRS, which pushes the FAC and parses the next value in the for-statement. The stack, at the point of the parse, looks like:

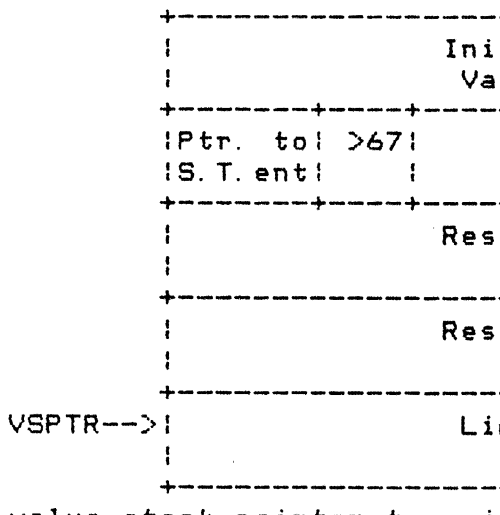


When the parse of the initial value is completed it is, as usual, returned in the FAC. It is then pushed onto the stack to yield a stack that looks like:

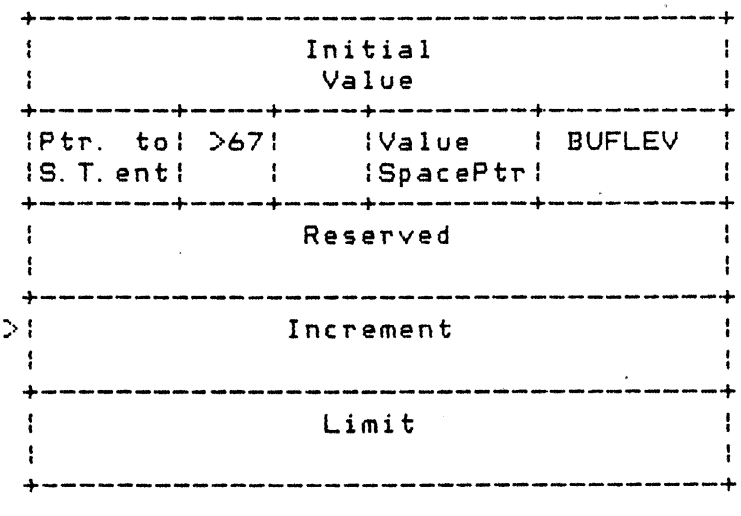


After some error checking is done, the to-value, or limit, is parsed and is returned in the FAC. At this point, the real stack manipulations begin in order to get the values into the correct positions in the stack block.

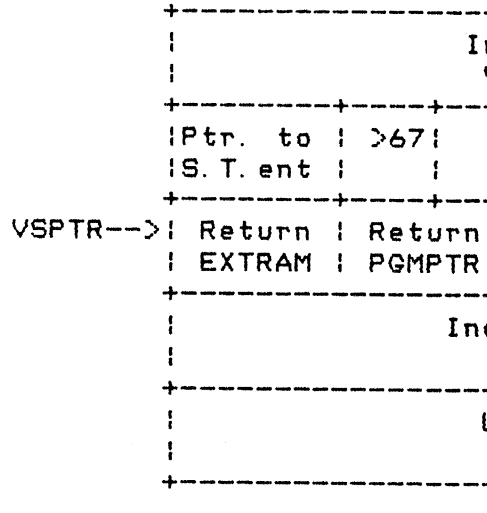
Forty (40) is subtracted from VSPTR to get it pointing at the value below the the for-block. The limit is now pushed on the stack and the stack now looks like:



Next, the stack pointer is reset to the top of the stack and the step value is parsed, if one exists, or a floating point one is put in the FAC as the default increment. Thirty-two (32) is now subtracted from the value stack pointer to point it at the limit again and the increment value is pushed into the stack-block yielding a stack like:



Now, the remaining stack entry is created by putting the current EXTRAM and PGMPTR in the FAC, decrementing the PGMPTR to be pushed so that the next-statement can resume in the proper place and the final entry is pushed yielding a stack block of:



Next, sixteen (16) is added to the value stack pointer to point it at the initial value. The initial value is then popped off of the stack into the FAC. The initial value is now assigned to the index variable by using ASSGNV. ASSGNV leaves the stack pointer pointing at the EXTRAM/PGMPTR entry.

Now a check must be made to determine if the for-loop should be executed at all Sixteen (16) is subtracted from the stack pointer to point it at the limit. The limit is now compared with the initial value (which is still in the FAC) to see if the for-loop should be executed by using SCOMPB. SCOMPB leaves the stack pointer pointing at the limit. A check is now made to see if the increment is positive or negative. This determines the direction that the initial/limit comparison should be interpreted. If the increment is positive, the initial value must be greater than or equal to the limit if the loop is to be executed.

If the loop is to be executed, thirty-two (32) is added to the value stack pointer to yield the final stack block and the body of the block is executed. If the loop is not to be executed, the remaining program text is searched for the matching next-statement. A count (R3) is kept of the number of for-statements encountered. It is initialized to one and is incremented for every for-statement encountered and decremented for every next-statement encountered until it reaches zero. We are guaranteed to find a next at the same level as the for-statement by the check that is made at prescan time. When the matching next-statement is encountered, the loop variable is checked to make sure that it is the same as the one used in the for-statement. If it is not the same, an error occurs. If it is the same execution resumes at the statement following the next statement. Please note that there is no need to flush the for-block off the stack because the value stack pointer was left at the bottom of the for-block by the comparison (SCOMPB) that was done in determining if the for-loop should be executed.

NEXT Statement

The next-statement is executed only in conjunction with the for-to-step statement which has the same index variable. The next-statement serves in either of two functions. One, it serves as the end of the for-next loop and causes the index variable to incremented/decremented appropriately and control to resume at the for-statement. Second, it can serve as the end of a skipped block in the event that a for-next loop is not executed. This second case has been described in connection with the for-statement.

The first thing that the next statement does is to pop the first entry of the for stack-block off the stack. If the entry popped off is not a for entry, then an error occurs. Next, the index variable is checked to see if the one in the next-statement matches the one in the for-block. If it does not then twenty-four (24) is subtracted (i.e., this for-block is discarded) form the value stack pointer and the stack is searched further for the matching for-block.

If the matching for stack-block is found, the next thing to be checked is the BUFLEV to make sure that the for-statement wasn't typed in and then lost before the next-statement was encountered. If the BUFLEV is doesn't match, an error has occurred.

Eight (8) is now subtracted from the value stack pointer to get it pointing at the increment and MOVFAC is used to get the index variable's value into the FAC and the increment is added to the value, leaving the result in the FAC and leaving the value stack pointer pointing at the limit. Twenty-four (24) is added to the value stack pointer to point it back at the top of the for stack-block and the new value is assigned to the index variable by using ASSGNV. ASSGNV leaves the value stack pointer pointing at the line number/line pointer entry.

Eight(8) is now subtracted from the value stack pointer to point at the limit and do a stack compare (SCOMPB), comparing the limit on the stack with the current value which has just been assigned to the index variable. We next check the increment value to see if it is negative or positive to determine which direction to interpret the comparison. If the value is outside of the bound, we simply continue as the value stack pointer due to the comparison (SCOMPB) that was done, effectively discarding the for-block. If we are within the bound, we add thirty-two (32) to the value stack pointer to retain the for-block on the stack. We then pick up the old line number table pointer (EXTRAM) and the pointer within the line (PGMPTR), set them up properly and go to CONT to resume execution at the beginning of the for-loop.

4.3.6.3.5 IF-THEN-ELSE

Execution of the if-then-else-statement consists of two parts. First, the expression of the if-clause is evaluated and second, the correct section of code to execute is selected and control flows to it. This can either be a line number to go to or one or more statements to execute. The expression in the if-clause is expected to produce a numeric result which can be either zero or non-zero. Zero is treated as a false value and any other value is treated as a true value. The logical operators described in section 4.3.7.6 produce values of either zero or negative one.

After the expression has been evaluated the if-statement handler checks for a 'then' token. If it is present, the value of the expression is checked to see if it is true and, if it is, a check is made to see if a line number is present. If one is, a dummy 'on' index is loaded (CLR R3, see sections 4.3.6.3.1 and 4.3.6.3.3.) and control flows to the common code used by the goto and gosub-statements. If a line number is not present a check is made to see if the end-of-line has been reached. If so, an error occurs. If not, it is assumed that a statement follows and the current character (R8) is loaded with the statement-separator token and the token pointer (PGMPTR) is decremented by one to fake out the parser into thinking that it has just encountered a statement-separator and must follow through to the next statement. Control then flows to the continue routine which will cause the next statement to be executed. Note that when the statements in the then-clause have been executed, an end-of-line, a tail-remark or an else-clause must be encountered. CONT has been set up so that it can handle an else in the same fashion as an end-of-line.

If the result of the expression is false when it is evaluated, an else-clause is searched for. Note that a counter is kept of the number of if-tokens and else-tokens encountered so that the else-clause which matches the then-clause is executed and not an intermediary else-clause which belongs to an if-statement which is part of the then-clause. Searching for the matching else involves searching, token by token, down the line for an else-token or until the end-of-line is reached. If an else-clause is present, control flows into the code described above which handles the execution of the then-clause.

4.3.6.3.6 CALL

The execution of the call-statement is one of the more complex in the interpreter. The problems arrise because of BASIC subprograms needing to have parameters passed to them. This section describes how the call statement is executed for both BASIC subprograms and for GPL subprograms.

Initially the subprogram name is read and the subprogram name symbol table is searched to find the subprogram name. If the subprogram is not found then control flows to the code written in GPL which tries to call the subprogram by calling the link routine of the monitor which will give an error if the program cannot be found. If the program is found, the in-use flag is checked to make sure that an illegal invocation of the subprogram is not about to occur. If a reinvocation is attempted an error occurs. If the subprogram is written in Assembly Language control passes into it directly, assuming that the subprogram knows how to handle any parameters that may be passed to it. The following information assumes that the subprogram is written in BASIC.

A check is made to see if there are any arguments. If there are arguments the following process is used (generally). It is assumed that each argument is a pass-by-reference until something is encountered that says it definitely is a pass-by-value. In the case that the argument is a pass-by-value it is treated as an expression and is evaluated accordingly and then assigned to the formal parameter. If the argument is determined to be a pass-by-reference it is checked to see if it has already been passed once as a parameter into the current calling program. If so, the symbol table entry is used to get back to the actual symbol table entry. The appropriate flags are then set in the formal parameter's symbol table entry and its value space is set as a reference field.

When all of the arguments have been evaluated and assigned to the parameters the stack entry (see section 4.3.3.1) has been built, execution proceeds into the subprogram.

Parameter Passing Parameter passage is the major feature of subprograms and there are some very strict conventions which are followed in order to properly pass the arguments into the subprogram.

Initially, a check is made to see if there are any arguments supplied in the call-statement. If there are not any, control proceeds directly into the code which builds the stack entry, prepares the interpreter for executing the subprogram and proceeds into the subprogram. Note that a check is made to make sure that the number of arguments matches the number of parameters.

When an argument is encountered in a call-statement it is assumed to be a pass-by-reference until it can be determined that it definitely is a pass-by-value. Pass-by-value can be determined when the argument is enclosed in parentheses or is an expression, in which case there is no chance that a result could be returned to the calling routine, as there is no variable to receive it. When it is determined that an argument is being passed by value, the symbol or expression is evaluated in the

normal manner. The formal parameter list in the sub-statement is then scanned to pick up the corresponding formal parameter, the modes of the argument and parameter are checked for a match, and, if they match, the value is assigned to the formal parameter.

The case of pass-by-reference is much more complicated than passing an argument by value. When an argument is determined to be a pass-by-reference many things must be checked before the formal parameter may receive the symbol referred to. The argument must be checked to see if it was passed by reference from another program segment into the one that is currently making a call. If it is already a pass-by-reference variable, it must be indirected through to get back to the actual variable. Note that there will never be more than one level of indirection to pass through since this convention is used at every level of subprogram activation. If an array is being passed by reference the number of dimensions in the argument must be the same as the number of dimensions in the parameter. If the number of dimensions don't match an error occurs. Once the actual variable has been determined, the formal parameter, corresponding to this argument, is picked up from the sub-statement and the value space is set to point to the actual's symbol table value space. The shared flag bit is then set appropriately to complete the formal's symbol table entry.

When all of the arguments have been evaluated, the stack entry is built and control is transferred into the subprogram.

4.3.6.3.7 SUBEXIT and SUBEND

When a subend-statement or subexit-statement is executed the stack is flushed back to the subprogram entry (see section 4.3.3.1), by using the interpreter variable, LSUBP, getting rid of any for-blocks, gosub-blocks, etc. from the stack. This leaves the stack in the same state as it was when the subprogram was invoked. The information in the subprogram-block is then used to restore the global interpreter variables and to restore the previous symbol table. Execution is then continued with the statement following the call-statement which caused the subprogram to be invoked.

4.3.6.4 Program Termination

A BASIC program may be terminated by any one of four methods. First, and foremost, execution may be terminated by BASIC having executed the last statement in the program. In order to interrupt execution without running off the end of the program the stop and end-statements are provided. Finally, as an aid in debugging BASIC programs, breakpoints and the

break-statement may be use to halt execution with the option of resuming execution at exact point where it left off.

4.3.6.4.1 Normal

Normal completion of a BASIC program occurs when the last statement in the program has been executed. This is detected by the parser when the movable line number table pointer (EXTRAM) reaches, and passes, the end of the line number table (STLN). This condition is checked everytime a statement has completed execution and the line table pointer (EXTRAM) is decremented by four. When it is determined that EXTRAM has passed STLN (has a lower address) the interpreter returns to top-level in the same manner it does when an imperative statement has been completed (see section 4.3.1).

4.3.6.4.2 STOP and END

The stop and end-statements are treated exactly the same in this BASIC. Neither is required in a BASIC program however if one is present it is executed in the following manner. The address in the statement table causes control to flow into the code which is used to return to the top-level in GPL. The code loads up the GPL address of the statement following the XML EXECG statement which caused the program to be executed and returns to the GPL interpreter with that as the GPL 'program counter'. In the GPL code, the statement following the EXEC instruction is a case statement which checks for what type of statement ending is taking place. In this case it is a normal end of execution and control flows into the top-level routine of the interpreter.

4.3.6.4.3 Breakpoints

Breakpoints are provided in this BASIC to allow the user a strong debugging facility. Breakpoints can occur in three different ways. First, a breakpoint can be set by using the break-command of BASIC. Second, a breakpoint can be taken by the execution of a break-command without a line number, and finally a breakpoint can be taken by the user depressing the shift-C key on the keyboard. Section 5.1 describes how breakpoints are set and what happens when a breakpoint is taken.

4.3.7 Functions and Operators

This section describes how all of the functions and operators contained in the Product 359 BASIC are handled. This

includes all of the arithmetic functions and operators, string functions and operators, relational operators and how user-defined functions are executed.

4.3.7.1 Arithmetic Operators

The arithmetic operators can be divided into two categories, the LEDs and the NUDs. The LEDs are all of the binary operators (+,-,*,^) and the nuds are the unary operators (+ and -).

Execution of the unary (NUD) operators are very simple and a small piece of common code used for the two. Each of the nud handlers does a parse to pick up the operand. The minus operator then does a negation of the value received and then the common code is executed. The common code merely checks to be sure that the value parsed is a numeric value and if it is an B @CONT instruction is executed, otherwise an error occurs.

Execution of all of the binary (LED) operators are very similar and common code is utilized to handle the common portions of each operator. Each of the operators does a parse to a particular level.

Each of the operators, plus, minus, multiply, divide and involution push the first operand onto the stack and then do a parse to pick up the second operand. Both plus and minus do a parse to the level of the minus. This insures that both plus and minus have the same precedence. Similarly, both multiply and divide parse to the level of the divide. This insures that both multiply and divide have the same precedence. After the parse for each operator has been done common code is entered to complete the operation. The common code checks to make sure that each of the arguments are numeric and then calls the appropriate floating point routine to complete the operation. After the floating point operation has been completed, a check is made so that an warning condition can be indicated in the case of an overflow, and then a branch to CONT is made to return control to the parser.

4.3.7.2 Arithmetic Functions

Included in the Product 359 BASIC interpreter are fifteen arithmetic functions. These may be divided roughly into two different types, trignometric and other functions. Included in the trignometric functions are the arctangent (ATN), cosine (COS), sine (SIN) and tangent (TAN). The remaining functions include the absolute value (ABS), end-of-file (EOF), exponential (EXP), integer (INT), natural-logarithm (LOG), maximum (MAX), minimum (MIN), pi (PI), natural-logarithm (LOG), random number

(RND), signum (SGN), and the square root (SQR) function. All of the nud handlers for the arithmetic functions are written in 9900 assembly language except for EOF, MAX, MIN, PI, and RND functions which are written in GPL.

4.3.7.2.1 Trignometric Functions

All of the trigmonmeric functions are handled in exactly the same way by the interpreter. Each of the NUD handlers stores the address of the function evaluator (address in TRINSIC) into R2 and then proceeds into a section of code entitled COMMON. COMMON checks to make sure that there is enough room on both the subroutine and value stacks for the floating point operations to take place. If there is not enough room an error occurs. COMMON then places the address of the correct routine (R2) onto the top of the subroutine stack, so that it can do a parse to pick up the argument. It then restores the routine address to R2, verifies that the argument parsed is a numeric and then does a BL *R2 to evaluate the function. When the function returns, COMMON then checks for any errors or warnings, handles them appropriately and, if it is able to continue, executes a B @CONT instruction to return to the parser.

The algorithms used to evaluate the trignometric functions evaluate a set of polynomials to approximate the correct answers. The algorithms will not be described in detail here, as they are common algorithms. Information on the algorithms and their origins may be found in the TRINSIC listing of BASIC.

4.3.7.2.2 Other Arithmetic Functions

Of the remaining arithmetic functions included in the Product 359 BASIC, the exponential (EXP), natural logarithmic (LOG), and the square root (SQR) functions are handled in exactly the same manner as the trignometric functions described in the previous section. As with the trignometric functions, the algorithms used to evaluate these functions will not be described here.

The greatest integer (INT) function is evaluated by the GRINT function but is called directly from the BASIC integer code and does not go through COMMON, as the other functions which are evaluated by routines in the intrinsic package.

The maximum (MAX) and minimum (MIN) functions are evaluated in exactly the same manner, except that each has its beginning which determines which way to interpret the comparison and then common code is executed which does the comparison and puts the correct value into the FAC. In executing each of the functions, common code is used which checks for the correct syntax, parses

the two arguments, compares them and returns the condition of the comparison to the two front-end pieces of code. The front-end code then interprets the comparison and either leaves the second argument in the FAC or moves the first argument from ARG to the FAC.

Execution of the pi (PI) function involves simply placing the constant value of pi into the FAC and executing an XML CONT instruction.

The random number function (RND) generates the next number in a pseudo-random number sequence.

The signum (SGN) function is a very simple to execute. It first parses its argument and verifies that the result that it gets is a numeric. It then checks to see if a value of zero has been returned and if it has, then SGN returns that to the parser. If a non-zero value has been generated, SGN uses a GPL TBR instruction to see if the number is positive or negative. It then does a MOVE instruction to place a floating point one into the FAC and if the argument was negative, the floating point one is negated to indicate that the argument was less than zero. When all this is completed, an XML CONT instruction is executed to return control to the parser.

The end-of-file (EOF) function nud handler is contained in the FLMGR assembly. The function is executed by first parsing the argument (which is the logical unit number) and finding the corresponding PAB in the I/O chain (see Home Computer File Management Specification). It then loads the I/O op code for status (09) into the PAB and calls the DSR to determine the status of the file or device. After the DSR returns, a floating point one is loaded into the FAC. A check is made of the returned status and if the file/device is a a physical end-of-file, the floating one in the FAC is negated. If the file/device is not at an end-of-file, the FAC is cleared. After the correct number is in the FAC, an XML CONT instruction is executed to return control to the parser.

4.3.7.3 String Operators

The only string operator supported by the Product 359 BASIC is that of concatenation (&). The concatenation operator is used to concatenate two strings together into one string.

4.3.7.3.1 Concatenation

The LED handler for concatenation, as are all of the string functions, is written in GPL and is contained in the EXEC assembly of BASIC.

Getting to the LED handler for concatenation is handled directly by accessing the tables of the parser.

To execute the concatenation operation, the left-hand operand is pushed on the stack and the right-hand argument is parsed. The lengths of the two strings are then added together. If the length of the result string would be greater than the implementation capacity of 255, then 255 is set as the length for the result string and a truncation warning occurs. The system get-string routine, GETSTR, is called to allocate a string for the result. The left-hand argument is then copied into the result string. As much of the right-hand string as is possible, depending upon truncation, is the copied into the result string, completing the concatenation operation. Control then returns to the parser via the XML CONT instruction.

4.3.7.4 String Functions

This section will deal with the string functions ASCII (ASC), character (CHR$), length (LEN), position (POS), repeat (RPT$), segment (SEG$), string (STR$) and value (VAL). Although the functions ASC, LEN, POS and VAL all return numeric values, they are being discussed in this section because they take string arguments and are four of the primary means of operating on strings. All of the NUD handlers for the string operations are written in GPL and are contained in the EXEC assembly of BASIC.

The ASCII function (ASC) returns the ASCII value, in decimal, of the first character of the string argument supplied to the function. After parsing the argument, ASC takes the first character of the string (null strings cause an error), puts it into ARG and then jumps into the LEN code to take advantage of common code to convert the character to its ASCII number.

The character (CHR$) function takes, as its argument, an integer value, which is converted into the one-character string containing the character specified by the ordinal position within the ASCII collating sequence. Execution of the CHR$ function is done by parsing the argument and converting it into an integer. A one-character string is the created, the integer value, which is the ASCII value, is copied into the string and a XML CONT instruction is executed to complete the function.

The string length (LEN) function is used to find the number of characters contained in the string specified by the argument. LEN parses the argument and takes the length of the string, returned in the FAC entry, from FAC+6/7, and then converts it into a floating point point number and executes an XML CONT instruction to complete execution of the function.

The position (POS) function is used to find the character position of the first character of the first occurrence of one string within another beginning at a particular character position within the source string. Execution of the POS function involves parsing the three arguments, pushing the first two on the stack, as soon as they are parsed. When the third argument is parsed, it is converted to an integer to be used internally to use as the starting character position for the comparison. The two string arguments are popped off the stack, moving the second argument (first popped off) into ARG from FAC. If the source string is found to be null, then the value zero is returned as no match can be made. The character position which has been specified is then compared to the length of the source string and if it is greater, then a zero is returned by the function, since no string match can be found. If the match string is null, then the value of 1 is returned, indicating that a match occured in the first column, as the null string matches any string. The strings are then compared, character by character, in an intricate loop in an attempt to find a match for the second string within the source string. If a match is found, then the characer position of the first character matching is returned in the FAC. If no match is ever found then a value of zero is returned in the FAC. Note that the code in LEN is used to convert the integer position of the match into a floating point value.

The repeat (RPT$) function is used to create a string with n copies of first character of the source string. Execution of the RPT$ function involves the parsing of the arguments and then creating a string with the required number of characters. If the number of characters specified is negative or non-numeric, an error occurs. If the number of characters exceeds 255 it is truncated. After the result string has been created, the source string is then replicated the required number of times to produce the desired result.

The string segment (SEG$) function is used to extract one string from another. The arguments specify the source string, the character position within the source string from which the new string is to begin and the length of the new string. Execution of the SEG$ function involves the parsing of the first two arguments, pushing them on the stack and then parsing the third argument. After the third argument has been parsed, converted to an integer and saved, the first two arguments are popped off the stack, saving the second argument (the first pooped off the stack) before popping the first argument. A comparison is made to see if the character position specified is greater than the length of the string, and if so, the null string is returned. If the character position specified plus the number of characters asked for is greater than the length of the source string then the entire source string beginning with the specified position is taken otherwise the specified portion of the source string is taken. When the exact length of the string to be taken has been determined, a string of that length is

created by using GETSTR and the correct string is copied into the result string and a string FAC entry is built. When this has been completed, an XML CONT instruction is executed to return control to the parser.

The string function (STR$) is used to convert a number into its string equivalent. It does this by first parsing the argument and converting it into a string by using the Convert Number to String (CNS) routine. STR$ then gets rid of any leading spaces and then calls the string literal routine (LITSTR) to create a string, copy the converted number into the string and to build the FAC entry. After completion of this an XML CONT instruction is executed to return control to the parser.

The numeric value function (VAL) is used to convert the string representation of a numeric value into a number. It does this by first parsing the argument and then converting it. STR$ converts the argument by removing any leading or trailing blanks from the source string and then allocating a new string with the exact number of characters contained in the number portion of the source string plus one. The numeric portion of the source string is copied into the new string and a blank-space character is appended to the end of the string to indicate to the conversion routine where the string ends. The conversion routine, CSN, is then called to convert the string into a number. When CSN returns a check is made to verify that the string was converted and that an illegal argument was not passed and if there where no errors an XML CONT instruction is executed to return control to the parser.

4.3.7.5 User-Defined Functions

Execution of user-defined functions is one of the more complicated parts of the Product 359 BASIC interpreter. Note that this implementation supports only one-parameter, single-line user-defined functions.

When a reference to a user-defined function is encountered control flows to the UDF code contained in the EXEC assembly of BASIC. The UDF code is entered with FAC containing the pointer to the symbol table entry for the function definition and CHAT containing the character following the referenced name. The first thing that is done is that a parameter count is initialized to zero (FAC+7), assuming that no arguments are provided.

If CHAT contains a left parenthesis then it is assumed that an argument is being supplied and it must be parsed. The pointer to the symbol table entry in FAC is pushed on the value stack to save it while the argument is being parsed. After the argument has been parsed it is moved from the FAC to ARG so that the

symbol table pointer can be popped off of the value stack and the argument count is incremented to indicate that a parameter has been encountered.

The symbol table pointer and number of arguments are now saved in temporary variables so that they can be used later. VPUSH is called to reserve an entry on the stack for the UDF entry that is to be constructed. The parse result (or a dummy entry if no arguments are supplied) is then pushed onto the stack for safe keeping. The first byte of the function's symbol table is fetched to see if the number of arguments supplied matches the number of parameters needed by the function. If it does not match, an error occurs.

The UDF stack entry is now constructed by copying PGMPTR, the string flag bit of the symbol table entry, the symbol table pointer and the free-space pointer into FAC through FAC+7, respectively. Until the parameter entry has been made on the symbol table, the stack ID for a user-defined function is >70 so that if an error occurs in the ENTER routine, the first entry on the symbol table will not be deleted when the stack is cleaned up by the error handler. The entry is then pushed onto the stack below the parse result.

Now a symbol table entry must be constructed for either the parameter or a dummy entry to keep the global variable scoping correct. This second case prevents one user-defined function having a parameter which has the same name as a global variable from invoking another user-defined function which uses that global variable from getting the value of the parameter.

After the parameter entry has been made into the symbol table, the real UDF stack ID, >68, is put into the UDF stack entry so that if an error occurs during the execution of the function, the parameter entry will be removed from the stack. The symbol table link in the parameter's symbol table entry is changed so that global scoping is used for variables appearing in the UDF definition. Now the argument value is popped off of the stack and the value is assigned to the parameter. Note that if no parameter was provided, garbage is assigned to the dummy symbol table entry which does not affect anything since the entry has a zero-length name and cannot ever be used for anything, except to give global values.

After the parameter value has been assigned to the parameter, the function definition is actually parsed to get the value for the function. When the parser returns to the UDF code, a check is made to be sure that the value returned has the same type as the function, i.e. a string function produces a string result and a numeric function returns a numeric result. If the type matches correctly then the parameter entry is delinked from the symbol table, the stack entry is retrieved to restore the old symbol table, free-space and program pointers are restored to resume execution at the point where the function was

referenced and then an XML CONT instruction is executed to return control to the parser.

4.3.7.6 Relational Operators

The relational operators, equal, not-equal, less-than, less-than-or-equal, greater-than, and greater-than-or-equal are handled by a common piece of code in the PARSE assembly of BASIC.

When one of the relational operators is encountered a value is assigned to the operation, 0 through 5 (assigned in the list above, with 0 being equal and 5 being greater-than-or-equal). This value is then used after a comparison of the two operands has been made to determine how the comparison is to be interpreted.

The comparison of the two operands is done by using the floating point routine SCOMP for numeric comparisons or by using special code for comparing two strings which is contained in the relational code. When comparing strings, the strings are compared character-by-character until all of the characters match, a character does not match, or all of the characters have matched but one string still has more characters. The three cases just described result in the strings being considered equal, not-equal or one string being considered greater-than the other one, respectively. In the third case, that all of the characters have been compared and are equal, and one string is longer, the longer string is considered to be greater than the shorter one.

After the comparison, either string or numeric, has been completed, then the number assigned to the comparison type (0 through 5, above) is used as an index into a small branch table to cause control to branch into a series of conditional jump-instructions which cause a 0 (false comparison) or a minus one (true comparison) to be loaded into the FAC. After this has been completed, a B @CONT is executed to return control to the parser.

4.3.7.7 Boolean Operators

The boolean operators, AND, OR and XOR, are all handled similarly as they are all LEDs. The operator, NOT, is handled a little differently as it is a NUD. All four of the operators are handled in the NUD assembly of BASIC.

In evaluating the three LEDs, AND, OR and XOR, the two arguments involved are converted into integers and the appropriate 9900 Assembly Language instruction is used to

perform the operation on the two integer operands. A SZC (Set Zeros Corresponding) instruction is used to evaluate the AND operation. A SOC (Set Ones Corresponding) instruction is used to evaluate the OR operation. An XOR (eXclusive OR) instruction is used to evaluate the XOR operation. Each of the operations is completed by converting the integer result to a floating-point value and returning to the parser.

The NOT operation is evaluated by parsing the argument, converting it into an integer, inverting it, and converting it back into a floating-point value. The 9900 INV (INVert) instruction is used to do the evaluation.

4.3.8 Error Handling

A significant portion of the Product 359 BASIC interpreter is devoted to the detection, reporting and handling of errors. The interpreter has many checks for syntax, illegal statements, memory overflow and semantic errors. Errors are detected as soon as possible once a program is entered into memory. Any errors which can be detected when a statement is entered are reported immediately. A number of errors which really would not occur until a program is executed are detected during the static scan of the program to report them as soon as possible. The following three sections describe what actions are taken by the interpreter when an error is detected and how the special ON ERROR and ON WARNING features of the interpreter are implemented.

4.3.8.1 Detection

Errors may be detected in any number of ways but once an error is detected the interpreter handles it in a specific manner. The error handling routine of the interpreter is entitled ERR$$ and is located in the PSCAN assembly of BASIC.

When an error is detected ERR$$ is called by using a CALL ERR$$ statement followed by one data byte which contains an error number which is actually an index into a table which is maintained in the error routine. A sample calling sequence is:

```
MSGSNM EQU  15
         :
         :
       CALL ERR$$              Issue the error message
       DATA MSGSNM             *STRING-NUMBER MISMATCH
```

The error message, STRING-NUMBER MISMATCH is contained in the EDIT assembly of BASIC and its address is contained in the error table of the error routine. MSGSNM is an equate for the

15th entry in the error table (15 is used purely for example here) which contains the error code and error severity reported by the CALL ERR subprogram and the address of the message text which can be displayed on the screen. Note that the internal number (index into the error table) does not necessarily match the error number which is reported to the user. When control reaches the error routine, it does a GPL FETCH instruction to get the index into the error table.

The preceeding discussion assumed that the error was detected by a Graphics Language portion of the interpreter. When an error is detected within the assembly language portions of the interpreter, a value is placed in the variable, ERRCOD, and control returns to GPL at the CASE statement following the XML EXEC instruction which started the statement or program executing. The case statement sends control to a small error routine in the EXEC assembly which then does another CASE statement to send control to a statement which calls the error routine with the appropriate error index following the call-statement.

4.3.8.2 Reporting

All error reporting is handled by the error routine, ERR$$, which is contained in the PSCAN assembly of BASIC. The error routine first fetches the one byte of the error table index so it can pick up the error number, severity and message address from the error table. It then checks to see if the symbol table pointer is pointing into the crunch buffer (UDFs can cause this) and if it is, it is restored to the correct place. It then fetches the information, described previously, from the error table.

If the severity of the error is zero, which has been defined to be only a system message is being displayed and not an error, the error-print routine, ERPRNT, is called to get the message from the GROM and display it on the screen. When control returns from the error-print routine, a check is made to see if the message which has just been displayed indicated that BASIC is going through an intialization procedure (displaying the opening message). If so, CLSALL is called to close any open files. The stack is then cleaned up and control returns to top-level.

If the severity of the error was not zero, which indicates that an actual error has occurred, a check is made to see if the ON ERROR condition has been set. If it has not, the trace-back routine, TRACBK, is called to provide trace-back of errors which have occurred in executing a user-defined function or an error which has occurred in execution of a BASIC subprogram. If the error has occurred in either one of these structures, the stack and symbol table is used to display the original error message

and then trace back through the levels of function and / or subprogram invocation to the main program to indicate exactly when and how the error occurred. If the error did not occur within the execution of one of these structures, TRACBK returns with the condition reset to indicate that the normal error-print routine needs to be called. If this is the case the procedure used is identical to when a message-only call to the error routine is made, as described above; the error-print routine is called, the stack is cleaned up and control is returned to top-level.

In the case that ON ERROR has been armed (by execution of an on-error-statement), the routine, CLEAN, is called to clean up the stack and the error stack entry is built using the information obtained from the error table. If there is room on the stack, the entry is pushed on it. If not, the original error message is displayed and another error message is displayed to indicate that the error recovery routine could not function. If this is the case, control returns to top-level. If there is room on the stack, the error entry is pushed, the saved line number table pointer is placed in EXTRAM (moved from VDP where it was saved by the execution of the on-error-statement), the VDP location is cleared to indicate that the ON ERROR has been taken and can't be taken again, and control is returned to the parser via the execution of an XML CONTIN which will cause execution of the program to resume with the line specified by the on-error line number.

4.3.8.3 Warnings

Warnings issued by the interpreter are handled in much the same way as errors, except that execution normally continues after the message has been displayed. The only warnings which can occur when a program is executing are: a NUMERIC OVERFLOW when a floating point operation has taken place and the machine's capacity has been exceeded; an INPUT ERROR when the wrong number of input items has been entered or the type of the items entered does not match the type of the variables into which the value is to go; and, a string truncation warning whenever a string has to be truncated because it exceeds the 255 allowable characters.

The warning routine, WARN$$, is called in the same manner as the error routine, ERR$$, with the index into the error table following the call statement, as in the following example:

```
MSGNO  EQU  16
         :
         :
       CALL WARN$$              Issue a warning message
       DATA MSGNO               * WARNING: NUMERIC OVERFLOW
```

The warning routine first fetches the index into the error table and then fetches the information contained there. It then checks the warning print-bit to see if the on-warning condition indicates that the message should be displayed. If it should, the screen is scrolled, the message, * WARNING, is displayed and the trace-back routine is called to possibly trace back the error if it occurred in the execution of a user-defined function or a BASIC subprogram. If so, TRACBK issues the messages and returns. If the error did not occur in one of the two cases, the error-print routine is called to display the message. The warning-stop-bit is then checked to see if execution is to continue. If execution should continue, an RTN instruction is executed to return to the caller. If the warning condition indicates that the program is to stop execution upon the detection of a warning, the stack is cleaned up and control returns to top-level.

4.3.8.4 On Error Statement

The on-error-statement is used to change the on-error condition from the default case to the special case and vice-versa. When an on-error-statement has been encountered by the ON statement code of the parser control flows to the ONERR routine contained in the EXEC assembly of BASIC. ONERR checks to see if a line number is present. If it is, the line-number table is searched to find the line and if it exists, the pointer within the table to the line specified is stored in the VDP RAM location which has been reserved for it. If the on-error condition being specified is to stop when an error has been detected, the special location in VDP RAM is cleared to indicate that the default error handling conditions exist(displaying the message and aborting). A check is then made to be sure that the statement ended correctly and control then returns to the parser.

4.3.8.5 On Warning Statement

The on-warning statement is handled in much the same way as the on-error statement except that the keywords, PRINT, STOP and NEXT are expected. If the keyword, STOP, is encountered, the print-bit is cleared to indicate that any messages are to be displayed, and the stop-bit is set to indicate that execution is to stop if a warning occurs. If the keyword, PRINT, is encountered, both the print-bit and stop-bit are cleared to indicate that any warning messages are to be displayed an execution is to resume after doing such. This is the default condition. If the keyword, NEXT, is encountered, the print-bit is set to indicate that warnings are not to be displayed, and the stop-bit is cleared to indicate that execution is to

continue, without displaying any message and without interrupting the flow of the program.

5.0 Debugging Aids

The Product 359 BASIC includes two aides for use in debugging BASIC programs, breakpoints and tracing. Breakpoints allow the user to cause execution to proceed as normal until a certain statement is about to be executed and execution is interrupted to allow the user to look at and assign values to variables and then resume execution. Tracing a program causes the line numbers of each line to be displayed on the screen before each statement is executed, allowing the user to see exactly how control is flowing through the program.

5.1 Breakpoints

Breakpoints in the Product 359 BASIC may be caused in one of three ways. First, the shift-C key on the keyboard may be pressed, halting execution. Second, a break-statement with no arguments may be executed. Third, a breakpoint may have been set by a break-statement and have been encountered. Each of these types is discussed here or in the following section.

Whenever the parser is about to begin execution of a new statement, it checks to see if the shift-C key is depressed on the keyboard. If it has been depressed, a break is taken by loading the break-flag code into ERRCOD and returning to GPL. The GPL code causes the screen to be scrolled and the breakpoint message is displayed on the screen, as well as the current line number. The current line number table pointer (EXTRAM) and line pointer (PGMPTR) are saved in the VDP RAM in special locations to indicate that a continue statement can be executed if it is entered by the user. The default character set is then restored and control returns to top-level to await a command.

The other two types of breakpoints are intimately tied up with the execution of the break-statement and are described in the following section.

5.1.1 BREAK

The break-statement serves two functions in the Product 359 BASIC. First, it can be used to set breakpoints at particular lines in a program and, second, it can be used in a program to cause a breakpoint to occur when it is executed (break-statement without any line numbers specified).

In order to set a breakpoint at a particular line in a BASIC program, the most-significant bit of the most-significant byte of the line number in the line number table is set to a

one. When the break-statement picks up a line number from the line-list, the line number table is searched to locate the line specified. If the line is not contained in the program then a warning message is issued, but, the rest of the line-list is searched, setting any other legal breakpoints. Once the line has been found the most-significant bit of the line number is set to a one to set the breakpoint.

This method is used because the line numbers are restricted to a maximum value of 32767 and so the most-significant bit can never be set to a one. It would be more efficient to set the most-significant bit of the line pointer to a one but with the addition of support for the expansion RAM peripheral and its 64K address space this method will not work. Thus, when the parser is ready to execute a new line it must first pick up the line number to check for a breakpoint. If the line number has the most-significant bit set to a one, the parser determines that a breakpoint has been set on that line and control flows to the breakpoint handler in exactly the same manner as when the shift-C key is depressed on the keyboard (described in the previous section).

When a break-statement without a line number list is encountered, it is interpreted as being an immediate breakpoint and control flows from the break-statement handler to the breakpoint code to indicate that a breakpoint has been taken. Before control reaches the breakpoint handler, the current line number table pointer and text pointer are saved so that if a continue-command is entered, the statement following the break-statement is where execution will resume.

5.1.2 UNBREAK

The unbreak-statement can be used to selectively remove or totally remove all breakpoints. When the unbreak-statement is executed without any line number specified, it goes through the entire program in memory and resets the top bit of all of the line numbers in the line-number table. This clears any and all breakpoints that might have been set. It makes no difference that a breakpoint is not set at a particular line since resetting the top bit of the number will make no changes to it if the bit is already reset.

If a line-list is specified, then the program is searched for each line, and the top bit of each line number is reset for each line listed. If a line is listed which does not exist, a warning condition occurs, but the remainder of the line list is processed.

5.1.3 On Break Statement

When an on-break-statement has been encountered by the ON statement code, control flows to the ONBRK code contained in the EXEC assembly of BASIC. One of the keywords, NEXT or STOP, is then expected. If the keyword, STOP, is encountered, the on-break bit is reset to indicate that, if a break-key or breakpoint is encountered, it is to be abided by and the breakpoint is to be taken. If the keyword, NEXT, is encountered, the on-break bit is set to indicate that, if a break-key or breakpoint is encountered, it is to be ignored and execution is to continue as if the break-key or breakpoint was never encountered.

5.2 CONTINUE

The code to execute the continue-command is contained in the EDIT assembly of BASIC, as it is a command and not an executable statement. When the continue command is executed, a check is made to see if a breakpoint has been taken. If one has (EXTRAM and PGMPTR were saved in the VDP by the breakpoint handler) been taken, the program flag is swithched back to program mode, continue is disabled (the special VDP RAM continue words are cleared), and an XML CONTIN instruction is executed to return control to the parser. If a breakpoint has not been taken, an error message is issued to state that BASIC can't continue executing a program if it has not executed a breakpoint.

5.3 Tracing

The trace feature of the Product 359 BASIC is a very simple one to execute. The trace and untrace-commands to turn on and off, respectively, the trace feature are described in the following sections.

When a statement is about to be executed, the parser checks to see if trace-mode is turned on and, if it is, the trace routine is called to put the line number on the screen. When control reaches the trace handler, it calculates the current screen address to see if the trace information can be put on the current line or if the screen must be scrolled fist. If there is not enough space on the screen, it is scrolled and the screen address is initialized to the beginning of the bottom line of the screen. The less-than character is then displayed, followed by the line number which has been calculated, and finally followed by a greater-than sign to close out the displayed information. The screen address is updated to be after the displayed line number and an XML RTNB instruction is executed to return to the parser to execute the statement for which the line

number has just been displayed.

5.3.1 TRACE

Execution of the trace-statement involves simply setting the trace flag bit (bit 4) of the interpreter variable, FLAG, and executing an XML CONT instruction to return control to the parser.

5.3.2 UNTRACE

Execution of the untrace-statement involves simply clearing the trace flag bit (bit 4) of the interpreter variable, FLAG, and executing an XML CONT instruction to return control to the parser.

6.0 Expansion-RAM BASIC Program Support

A complete description of the support for the expansion RAM peripheral is contained in the Product 359 BASIC Interpreter Expansion RAM Peripheral Support Software Specification and will not be duplicated here. That document is considered to be part of this specification and read as such.

7.0 GPL Subprograms

This section describes the GPL subprograms contained in the Product 359 BASIC interpreter. The subprograms supported include: CLEAR, HCHAR, VCHAR, GCHAR, CHAR, KEY, SOUND, COLOR, SCREEN, VERSION, SPRITE, DELSPRITE, DISTANCE, COINC, POSITION, MAGNIFY, MOTION, LOCATE, PATTERN, SAY, and SPGET. The subprograms which have the same names as those in the 99/4 BASIC have been rewritten and included in the Product 359 package because of necessary changes to support multi-statement lines and changes to allow them to be invoked by the call-statement, as it has been rewritten to support BASIC subprograms. The subprograms contained in the 99/4 BASIC are not accessible from this BASIC and those in this BASIC are accessible from the 99/4 BASIC.

7.1 CLEAR

The clear-subprogram is used to clear the entire screen. Execution of the subprogram involves execution of a GPL "ALL" instruction to set all of the characters on the screen to spaces, initializing the screen column pointer to the third column and returning to the caller.

7.2 SOUND

Execution of the sound-subprogram involves parsing the arguments, building a sound list in the CPU RAM and then moving it into the VDP RAM and issuing an I/O call to the sound generators. The sound-subprogram begin by parsing the duration of the sound which is to be generated by this invocation of the subprogram. If the duration is negative, the current sound, if any, is terminated and the duration is made positive. It is then converted into 1/60's of a second and saved in a temporary location in the CPU RAM. Next, the frequency/attenuation pairs for the three sound generators (or for one or two sound gererators if that is all that is supplied) are parsed and placed in the table being built in the CPU RAM. If a negative frequency is encountered, the noise control byte in the table is loaded with the correct value. If a fourth frequency/attenuation pair is provided, it is treated as noise control and is loaded into the table at the appropriate location. After all of the arguments to the sound-subprogram have been parsed and the table for the sound generators has been built, the subprogram then waits until any previous sound is completed and then loads the new sound table out to the sound generators and issues an I/O call to start the sound generators producing the desired sound.

7.3 COLOR

The color-subprogram is used to change the foreground and background colors of particular character sets (portions of the entire character set) or to change the color of a particular sprite. The usage of the color-subprogram to change the color of a sprite is described in the Product 359 Sprite Specification and will not be described here.

Execution of the color-subprogram when changing the colors of a character set begins by parsing the character set number and converting it into an address into the character color table. The foreground and background colors are then parsed, verified to be in the correct range (1-16) and then are loaded into the color table at the address calculated from the character set specified.

7.4 SCREEN

The screen subprogram is used to change the background color of the screen. Execution of the subprogram involves insuring that the left parenthesis is present, parsing the color specification, converting it into an integer and loading VDP register 7 with the color specification and returning to the calling program.

7.5 CHAR

The char-subprogram is used to define special characters. This includes defining both new characters and redefining the standard character set which is contained in the interpreter. The characters in the character set are used for both the standard pattern generation and for the sprite definitions. A complete description of the char-subprogram appears in the Product 359 Sprite Specification.

7.6 KEY

The key-subprogram is used to determine if a key is being depressed on a keyboard (either the console keyboard or a remote keyboard). The subprogram returns the keyboard status and the key code if a key is being depressed. Execution of the subprogram begins by parsing the keyboard unit which is to be scanned and then scanning the appropriate keyboard unit. The key-code assigned to the return variable will be either the ASCII representation of the key or zero if no key is depressed.

The value assigned to the status variable will be zero if no key is depressed, minus-one if the same key is depressed as when the last call to the key-subprogram was made, and one if a new key has been depressed since the last call to the key-subprogram was made.

7.7 VCHAR

THE vchar-subprogram is used to vertically repeat a character on the screen. The two arguments supplied to the subprogram indicate the position on the screen where the first character is to be displayed. The third argument specifies the ASCII character number of the character to be displayed on the screen. The fourth argument is optional and if it is present specifies the number of times the character is to be repeated vertically on the screen. The characters are displayed from the starting position, down the screen, wrapping around to the top of the screen when at the bottom of the screen and wrapping around to the upper left-hand corner when the lower right-hand corner is reached.

Execution of the subprogram begins by parsing the screen position arguments and initializing the screen postion to the correct position. It then parses the character number and saves it in a temporary. If the optional repeat count is present it is parsed, otherwise a default of one is used. Finally, a loop is entered which displays the character on the screen as many times as necessary. When the character has been displayed the required number of times, control returns to the calling program.

7.8 HCHAR

The hchar-subprogram is exactly the same as the vchar-subprogram except that the character is repeated horizontally on the screen, wrapping around to the following line when the end of one line is reached and wrapping around to the upper left-hand corner of the screen when the lower right-hand corner is reached. Execution of the subprogram is essentially the same as the execution of the vchar-subprogram and the reader is referred to the previous section.

7.9 GCHAR

The gchar-subprogram is used to read a character off of the screen at a specified location. Three arguments are necessary for the execution of the subprogram, the row and column positions to read and a return-variable to which the character number of the character occupying the position on the screen can

be assigned. Execution of subprogram involves parsing the screen position arguments and setting up the screen address accordingly and then reading the character off of the screen at that position. The character value is then converted into its floating point representation and is assigned to the return-variable, completing execution of the subprogram.

7.10 VERSION

Execution of the version-subprogram involves first insuring that the character following the name is a left parenthesis. If so, SYM and SMB are called to get the necessary information about the return variable and the FAC entry resulting from these two calls is pushed on the stack. Next, a check is made to be sure that a right parenthesis follows the variable name and if it is present, a floating point 100 is placed in the FAC and the ASSGNV routine is called to assign the value to the return variable, completing the execution of the subprogram. Control then returns to the calling program.

7.11 Sprite-access Subprograms

The sprite-access subprograms are described in the Product 359 Sprite Specification and will not be described here. The reader is referred to that specification, with the knowledge that it is considered to be a part of this specification.

7.12 Speech-access Subprograms

There are two speech-access subprograms provided in the Product 359 BASIC interpreter, SAY and SPGET. The say-subprogram is used to actually speak a word or string and the spget-subprogram is used to look up words in the speech library and return the actual speech data to the BASIC subprogram without actually speaking the word(s).

Execution of the say-subprogram involves first checking to see if the speech peripheral is present and if so making sure that a left parenthesis follows the subprogram name. The first argument is then parsed. If it is not a string argument, an error occurs. If the length of the string is non-zero, the speech DSR is called to look up the word in the speech dictionary and then actually speak it. Next, a comma is checked for and if it is present, the next argument is parsed, insured to be a string and not too long and the string is fed to the speech DSR for direct speaking of the string. If a comma is present after the second argument has been parsed, the subprogram loops back to the top and looks for another

word-string to look up and speak. If a comma is not present, a right parenthesis must be present and if it is the subprogram returns to the caller. Briefly, in pseudo-code, the say-subprogram looks like:

```
SAY     IF NOT LEFT-PARENTHESIS THEN ISSUE_ERROR
        REPEAT
          PARSE WORD_STRING
          IF NOT STRING_ARGUMENT THEN ISSUE_ERROR
          IF STRING_LENGTH NOT EQUAL ZERO THEN
            SPEAK_WORD
          IF COMMA THEN
            PARSE DIRECT_STRING
            IF NOT STRING_ARGUMENT THEN ISSUE_ERROR
            IF STRING TOO LONG THEN ISSUE_ERROR
            SPEAK_DIRECT_STRING
          IF RIGHT PARENTHESIS THEN RETURN
        UNTIL NOT COMMA
        ISSUE_ERROR
```

Execution of the spget-subprogram involves first insuring that a speech library is present. If one is not present, an error occurs. If one is present, the word or phrase for which the speech data is to be fetched is parsed. If the length of the string is non-zero, any timing characters which may appear at the beginning of the string are skipped over and if the string length is still non-zero the phrase or word is searched for in the speech library. If the data string for the speech data is too long an error occurs. Otherwise, a temporary string is created in the string space and the speech data is "copied" (not simple copy) into the temporary string. Finally, the speech data string (could be the null string because of checks made above) is assigned to the return variable for use by the BASIC program.

Appendix A - BASIC Keyword Table

Keyword	Internal Value	Keyword	Internal Value
ELSE	81	::	82
!	83	IF	84
GO	85	GOTO	86
GOSUB	87	RETURN	88
DEF	89	DIM	8A
END	8B	FOR	8C
LET	8D	BREAK	8E
UNBREAK	8F	TRACE	90
UNTRACE	91	INPUT	92
DATA	93	RESTORE	94
RANDOMIZE	95	NEXT	96
READ	97	STOP	98
DELETE	99	REM	9A
ON	9B	PRINT	9C
CALL	9D	OPTION	9E
OPEN	9F	CLOSE	A0
SUB	A1	DISPLAY	A2
IMAGE	A3	ACCEPT	A4
ERROR	A5	WARNING	A6
SUBEXIT	A7	SUBEND	A8
RUN	A9	LINPUT	AA
(LIBRARY)	AB	(REAL)	AC
(INTEGER)	AD	(SCRATCH)	AE
undefined	AF	THEN	B0
TO	B1	STEP	B2
,	B3	;	B4
:	B5	)	B6
(	B7	&	B8
undefined	B9	OR	BA
AND	BB	XOR	BC
NOT	BD	=	BE
<	BF	>	C0
+	C1	-	C2
*	C3	/	C4
	C5	undefined	C6
Quoted string	C7	Unquoted string	C8
Line number	C9	EOF	CA
ABS	CB	ATN	CC
COS	CD	EXP	CE
INT	CF	LOG	D0
SGN	D1	SIN	D2
SQR	D3	TAN	D4
LEN	D5	CHR$	D6
RND	D7	SEG$	D8
POS	D9	VAL	DA
STR$	DB	ASC	DC
PI	DD	REC	DE

Keyword	Internal Value	Keyword	Internal Value
MAX	DF	MIN	E0
RPT$	E1	(UPRC$)	E2
(STATUS)	E3	(TIME$)	E4
(DAT$)	E5	(INTG)	E6
(ALPHA)	E7	NUMERIC	E8
DIGIT	E9	UALPHA	EA
SIZE	EB	ALL	EC
USING	ED	BEEP	EE
ERASE	EF	AT	F0
BASE	F1	(TEMPORARY)	F2
(VARIABLE)	F3	(RELATIVE)	F4
INTERNAL	F5	(SEQUENTIAL)	F6
OUTPUT	F7	(UPDATE)	F8
(APPEND)	F9	FIXED	FA
(PERMANENT)	FB	TAB	FC
#	FD	VALIDATE	FE

The highest (FF) and lowest (80) values have not been assigned. The highest (FF) will never be assigned a value as all precedence testing by the parser uses the fact that it is not assigned and it is therefore considered an illegal value. Keywords in brackets have not actually been implemented in this BASIC, but have been assigned values in the anticipation of a later generations of personal computers using them. Please note that each keyword has a hexadecimal value with the most significant bit set. This condition is what actually differentiates the tokens from symbols.

Appendix B - GPL359 As A Debugging Aid

Experience with the TI 99/4 BASIC interpreter has shown that, by far, the most effective way to debug new sections of code in the interpreter is to use what is known as GPL10 on the 990/10. GPL10 is a software emulation of the 99/4. A special 990 SCI proc, GPL359, and an installed task were used to invoke the emulator. It has a customized GPL interpreter as well as several support modules which provide the interface between the emulator and the 990 operating system.

A slightly modified assembly language (99/4 ROM) code is link-editted in with the customized GPL interpreter and support routines to use the 990 debugging facilities. The modifications needed in the assembly language code are described in a later section.

When the GPL359 emulator is invoked (with the GPL359 SCI command on the 990) a block of memory is allocated which is divided up into the three remaining types of memory (CPU RAM, VDP RAM, and GROM) in the 99/4.

The CPU RAM is allocated first in the 990's memory, then the VDP RAM and finally the GROM. The SCI GPL359 command prompts for an object file which is the linked GPL object file that is to be debugged. This GPL object file must contain the system monitor, a patch file for the monitor (described in a later section) and the Product 359 GPL code.

Assembly Language

In order to use GPL359, a special link file was created for the link-editor. This file looks like:

```
TASK GPL359
LIBRARY      .SCI990.S$OBJECT
INCLUDE PPC2.P359.OBJ.MAIN
INCLUDE PPC2.P359.OBJ.CRT
INCLUDE PPC2.P359.OBJ.DEBUG
INCLUDE PPC2.P359.OBJ.FLTPT
INCLUDE PPC2.P359.OBJ.CSN
INCLUDE PPC2.P359.OBJ.CNS
INCLUDE PPC2.P359.OBJ.TRINSIC
INCLUDE PPC2.P359.OBJ.BASSUP
INCLUDE PPC2.P359.OBJ.STRING
INCLUDE PPC2.P359.OBJ.PARSE
INCLUDE PPC2.P359.OBJ.NUD
INCLUDE PPC2.P359.OBJ.CIF
INCLUDE PPC2.P359.OBJ.FORNEXT
INCLUDE PPC2.P359.OBJ.XML990
INCLUDE PPC2.P359.OBJ.SCROLL
INCLUDE PPC2.P359.OBJ.GREAD
INCLUDE PPC2.P359.OBJ.GWRITE
INCLUDE PPC2.P359.OBJ.DELREP
INCLUDE PPC2.P359.OBJ.MVDN
INCLUDE PPC2.P359.OBJ.MVUP
INCLUDE PPC2.P359.OBJ.VGWITE
INCLUDE PPC2.P359.OBJ.GVWITE
INCLUDE PPC2.P359.OBJ.GETPUT
INCLUDE PPC2.P359.OBJ.SUBPROG
INCLUDE PPC2.P359.OBJ.SCAN
INCLUDE PPC2.P359.OBJ.CRUNCH
INCLUDE PPC2.P359.OBJ.CPT
INCLUDE PPC2.P359.OBJ.GETNB
INCLUDE PPC2.P359.OBJ.SPEED
SEARCH
INCLUDE PPC2.P359.OBJ.EMULATOR
INCLUDE PPC2.P359.OBJ.EQUATE
END
```

MAIN, CRT, DEBUG, DATA, EMULATOR and the LIBRARY and SEARCH are required modules to get the special interpreter and to get the debug and 990 interface code.

In order for assembly language code which accesses the GROM, VDP RAM or expansion RAM to run properly, conditional assemblies are used. This is due to the differences in accessing the 990's continuous memory and GROM, VDP and expansion RAM on the 99/4.

Accessing GROM

The following is a typical example of how the GROM is accessed from assembly language:

```
MOVB @ADDR,@GRMWA(R13)     Write 1st byte of address
MOVB @ADDR+1,@GRMWA(R13)   Write 2nd byte of address
NOP                        Waste some time
MOVB *R13,@BYTE            Read a byte
```

The same access to the GROM is done in GPL359 by:

```
MOV  @ADDR,R14             Use R14 for common purpose
AI   R14,GROM              GROM offset in memory
MOVB *R14+,@BYTE           READ a byte
```

GRMWA and GROM need to be 'REFed' in the assemblies, as they are DEFed in the emulator.

Accessing VDP RAM

Accessing the VDP is more difficult than accessing the GROM in the real machine due to the need to write out the least significant byte of the address before the most significant byte of the address. Also, since the VDP can be both read from and written to another step must be added. The following is a typical example of reading from the VDP RAM:

```
SWPB R1                    Address is in R1
MOVB R1,*R15               Write 2nd byte of address
SWPB R1                    Swap back
MOVB R1,*R15               Write 1st byte of address
NOP                        Waste some time
MOVB @VDPRD,R2             Read the byte into R2
```

The same code written to read from GPL359's "VDP" area is written as :

```
MOV  R1,R14                Use R14 for the read
AI   R14,VRAM              Add offset into memory
MOVB *R14+,R2              Read the byte
```

Once again, VDPRD and VRAM must be DEFed in the emulator and must be REFed by the module using them.

In order to write to the VDP RAM a similar procedure to reading must be used. The following example demonstrates the reverse of the above example.

```
SWPB R1                    Address in R1
MOVB R1,*R15               Write 2nd byte of address
SWPB R1                    Swap to right order
ORI  R1,WRVDP              Or in the write enable
MOVB R1,*R15               Write 1st byte of address
NOP                        Waste some time
MOVB R2,@VDPWD             Write the data byte
```

The same example done for GPL359:

```
MOV  R1,R14          Use R14 for the read
AI   R14,VRAM        Add offset into memory
MOVB R2,*R14+        Write the data byte
```

Once again, WRVDP, VDPWD and VRAM must be REFed.

Accessing Expansion RAM

Since the expansion RAM is configured with a CPU-type interface accessing it on the 990 is essentially identical to accessing it on the 99/4 except that the simulated expansion RAM's memory offset must be added to the actual address in order to access the correct location in memory. The following example demonstrates how the expansion RAM is accessed on the 99/4 and on the 990.

```
MOV  @ADDR,R1        Fetch the address
MOVB *R1,R2          Read the byte
```

is used to access the expansion RAM on the 99/4 and

```
MOV  @ADDR,R1        Fetch the address
AI   R1,GRAM         Add expansion RAM offset
MOVB *R1,R2          Read the byte
```

is used to access the simulated expansion RAM on the 990.

GPL Code

The only thing that must be done to correctly execute GPL code on GPL359 is to include a file called MONPATCH in with the link of the GPL code. MONPATCH patches out some bytes of the monitor which the emulator considers to be illegal instructions because of some error checking is done. The bytes that must be patched out all deal with trying to write to the GPL interpreter workspace registers which GPL359 does not allow to be done from GPL code. These attempts to write to the workspace registers occur when the monitor is scanning the GROM area for GROM headers and when the linking routine, CPL, and the return routine, RPL, attempt to save and restore the interpreter's R13. Whenever the monitor is modified the patch file must also be modified to match the addresses of where the patched instructions are located. The locations of the patches may be found by looking at the source of the monitor and monpatch.

It should also be noted that none of the graphics available on the 99/4 are available on the 990, but, any characters put on

the screen of the 99/4 will appear on the 990's CRT in some form or another. A 99/4 console or simulator is necessary to display the correct characters on the screen.

Appendix C - Special GPL XMLs

In order to improve the speed of the interpreter, the most frequently used code has been written in 9900-code. This has necessitated the use of the GPL XML instruction to "call" a 9900-code subroutine. This interpreter utilizes two XML tables, yielding a total of 32 separate XMLs for speed-up and general usage purposes. The two XML tables, numbers 7 and 8, are contained in the XML359 module of the assembly language code and are located at locations >6010 and >6030, respectively, as set by the addresses defined within the 99/4 system.

Figure APP C.1 shows the contents of the first XML table.

Table 7 (>6010)

>70 - COMPCT	-	string garbage collector
>71 - GETSTR	-	system string allocator
>72 - MEMCHK	-	symbol table & PAB memory allocator
>73 - CNS	-	Convert Number to String
>74 - PARSE	-	begin a parse
>75 - CONT	-	continue a parse
>76 - EXEC	-	execute a BASIC program or line
>77 - VPUSH	-	push 8-byte entry on the stack
>78 - VPOP	-	pop an 8-byte entry off the stack
>79 - PGMCHR	-	fetch the next program character
>7A - SYM	-	find a symbol in the symbol table
>7B - SMB	-	evaluate symbol subscripts
>7C - ASSGNV	-	assign a value to a symbol
>7D - FBSYMB	-	search symbol table (for prescan)
>7E - SPEED	-	General speed-up routines
>7F - CRUNCH	-	crunch an input line

Figure APP C.1

The parse XML is special in that it must be followed by a data byte indicating the level to parse to. Therefore, a call to parse has the form of:

```
XML  PARSE        Parse a value
DATA RPAR$        Stop on ')' or higher
```

The speed XML is also special in that it is an XML that accesses three different routines and must be followed by at least one data byte to select the proper routine. The three routines are: 1) SYNCHK which checks for a syntactically-required token value and returns the program character following the required token or issues an error; 2) PARCOM which parses up to a comma, checks for a required comma and returns the program character following the comma else issues an error; 3) RANGE which converts a floating point number in the FAC into an integer and then verifies that it is within a certain range of values. These values ar specified by three data bytes following the XML and the RANGE selector. The form of the three XML routines accessed by the SPEED XML are:

```
XML   SPEED
DATA  SYNCHK        Select synchk (SYNCHK equated to 0)
DATA  LPAR$         A '(' is required

XML   SPEED
DATA  PARCOM        Select parcom (PARCOM equated to 1)

XML   SPEED
DATA  RANGE         Select range (RANGE equated to 2)
DATA  1,#300        Low value is 1, high value is 300
```

The CRUNCH XML also requires a selector field following the XML instruction to select the type of crunch being requested. There are two possible types of crunch, a normal input-line crunch of BASIC statements and a special crunch for the input-statement of BASIC to crunch data entered from the keyboard or from a display-type file. The CRUNCH XML call, therefore, has the two following forms:

```
XML   CRUNCH        Crunch a BASIC source line
DATA  0             Select normal crunch mode

XML   CRUNCH        Crunch the input data
DATA  1             Select data-crunch mode
```

Figure APP C.2 shows the contents of the second XML table.

Table 8 (>6030)

```
>80 - CIF      -  Convert Integer to Floating
>81 - CONTIN   -  continue after breakpoint
>82 - RTNB     -  return to 9900-code after call to GPL
>83 - SCROLL   -  scroll the screen
>84 - IO       -  general I/O speed-up routines
>85 - GREAD    -  read from expansion RAM to CPU RAM
>86 - GWRITE   -  write from CPU RAM to expansion RAM
>87 - DELREP   -  delete line text from program image
>88 - MVDN     -  move from low to high address in VDP/ERAM
>89 - MVUP     -  move from high to low address in ERAM
>8A - VGWITE   -  move from VDP to ERAM
>8B - GVWITE   -  move from ERAM to VDP
>8C - GREAD1   -  read from expansion RAM to CPU RAM
>8D - GWITE1   -  read from CPU RAM to expansion RAM
>8E - GDTECT   -  search for ERAM and enable page 0 at >7000
>7F - SCAN     -  prescan a program or line
```

Figure APP C.2

As with the XMLs in table 1 described above there are two XMLs in table 2 which have special calling sequences.

The I/O xml is used to access four different routines and the calls to IO have the following forms:

```
XML   IO
DATA  LLIST       Line list for list (LLIST equated to 0)

XML   IO
DATA  FILSPC      Fill record with fillers (FILSPC equated to 1)

XML   IO
DATA  CSTRIN      Copy string onto screen (CSTRIN equated to 2)

XML   IO
DATA  CLRGRM      Clear ERAM (CLRGRM equated to 3)
DATA  FAC         Address of 2-byte # of bytes to clear
DATA  FAC2        Address of address to begin clearing
```

The scan XML is used to speed up the prescanning of a program by having the main loop of the static scanner in 9900-code. There is a selector associated with the XML to select the condition upon which the routine is being called. The XML is called with the following forms:

```
XML   SCAN        Initial call to scan a program
DATA  0

XML   SCAN        Return to scan after call to GPL to handle
DATA  1                DIM, SUB, enter, etc.

XML   SCAN        Initial call to scan an imperative line
DATA  2
```