TI-99/8 HOME COMPUTER

BASIC INTERPRETER

Design Specification

Consumer Group
Mail Station 5890
2301 N. University
Lubbock, Texas 79414

TEXAS INSTRUMENTS
INCORPORATED

Date: August 3, 1983
Version 1.0

TABLE of CONTENTS

Paragraph Title

SECTION 5 INTERPRETER COMPONENTS

SECTION 6 BASIC STATEMENTS AND SUBPROGRAMS

LIST of TABLES

LIST of FIGURES

SECTION 1

INTRODUCTION

This document contains the design details and documentation of the TI-99/8 Home Computer BASIC language processor. This document is in a preliminary state and is subject to change. Section 3, Appendix A, and Appendix B have been reviewed and hopefully are technically correct. The rest of this specification is not guaranteed to be technically accurate.

1.1 Purpose

The information provided in this document is intended to provide a comprehensive documentation of how the TI-99/8 BASIC interpreter functions. This document can be referenced by persons maintaining, modifying, or using the interpreter on an 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.

1.2 Scope

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

This document also provides necessary information for people writing applications software in BASIC but needing to write some portions of their BASIC programs in Assembly Language or GPL.

SECTION 2

APPLICABLE DOCUMENTS

Home Computer BASIC Language Specification
(Revision 4.1 April 12, 1979)

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

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

File Management Specification for the TI-99/4 Home Computer
(Version 2.5, Revised February 25, 1983)

Software Development for the Texas Instruments Home Computer
(Released May 24, 1979)

TMS 9900 Microprocessor Data Manual
(Released December 1976)

Texas Instruments Graphics Programming Language Programmer's Guide
(Revised June 1, 1979)

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

SECTION 3

GENERAL DESCRIPTION

The BASIC interpreter included in the TI-99/8 console is an integral part of the system software included in the 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 TI-99/8's hardware. The interpreter provides several enhancements above the ANSI nucleus to access the color graphic capability and to access the sound generators contained in the machine.

3.1 Use of System Resources

This section describes how the interpreter utilizes the memory resources of the TI-99/8 console, including the RAM and ROM contained within the system. The interpreter resides in approximately 32K of ROM and approximately 18K of GROM. As much of the speed critical code as possible was put in the high speed ROM to increase the speed of the interpreter.

BASIC utilizes the CPU RAM for program storage, symbol table storage, Peripheral Access Blocks, string space, crunch buffer, and the floating-point stack. The CPU RAM is also used to maintain all of the necessary pointers and temporary variables involved with editing, prescanning, and executing a BASIC program.

BASIC utilizes the VDP RAM for screen I/O.

3.1.1 ROM Usage.

The ROM contained in the console contains many of the most frequently executed parts of the BASIC interpreter. The ROM code portion of the interpreter is contained in 19 separate assembly modules. Two of these modules are entitled PARSE and BASSUP. PARSE contains the statement dispatcher, the parser front-end and ending code, and some of the most frequently used NUD, LED, and statement handlers (see section 4.3). It also contains the CPUSH and CPOP routines which get values on and off the value stack. BASSUP contains several of the most frequently used support routines such as the symbol table search routine, symbol

assignment routine, and several small routines to access the VDP RAM.

In order to put together all of the ROM code for the entire system, each of the separate modules of the system must be included in a link-edit. The modules contained in the TI-99/8 console are:

```
INTSIM - The GPL interpreter
XOPS   - The memory mapper utilities
BIPHAS - The cassette device service routine
FLTPT  - The floating point package
CSN    - The convert string to number routine
SCREEN - The screen handler
NEWXML - The XML table
X2SUBS - The utility routines callable from GPL, or as an
         XOP/XML2 call
X3SUBS - The utility routines callable from GPL, or as an
         XOP/XML3 call
TRINSIC - The intrinsic functions
CNS    - The convert number to string routine
STRING - The string package
BASSUP - The BASIC support package
FLMGRSUP - The file manager support routines
PARSE  - The BASIC parser
FORNEXT - The FOR/NEXT command
SCAN99X - The prescan routine
SUBPRGX - The subprogram routines
FILES   -
BLWPTAB -
ASSMSUM -
```

Since the assembly language portions of the console can be link-edited there are no problems in putting all of the modules together, as there are with the GPL portions of the console.

3.1.2 GROM Usage.

The GROM portion of the interpreter contains part of the BASIC interpreter. The line input, crunching, program editing, and some of the statement NUD and LED handlers are contained in GROM. All input and output routines, and all of the GPL subprograms such as SOUND, COLOR, and KEY are located in GROM. The error messages' text as well as the error handling routine are also located in GROM code.

The GROM portion of the interpreter is contained in five separate assembly modules. These modules are entitled: EDIT, PSCAN, FMEX, and GSUB. These mnemonics describe the portions of

BASIC handled by each of these assemblies. EDIT handles mainly the editing, crunching, and top-level portions of BASIC. PSCAN contains the error handling routine as well as numerous subroutines such as the line input routine. EXEC contains all of the GPL portion of execution except the screen/file management portions which are contained in the FILEMGR assembly. GSUB contains all of the BASIC provided GPL subprograms, such as CLEAR, SOUND, and COLOR.

In order to include all the separate assemblies of the console together, a GPL object code linker is available. The linker is simply a file concatenator. There is no link-editing capability, so linkage from one assemblu to another is fairlu

In order to keep track of the amount of GROM used and which parts are used by which sections, a GROM map is kept. The GROM map reflects the organization and usage of the console GROMs. The following figure is the GROM map for the latest revision of the console GROM code.

TI-99/8 GROM MEMORY MAP
REVISION x
RELEASED xx/xx/xx
xx/xx/xx xx:xx

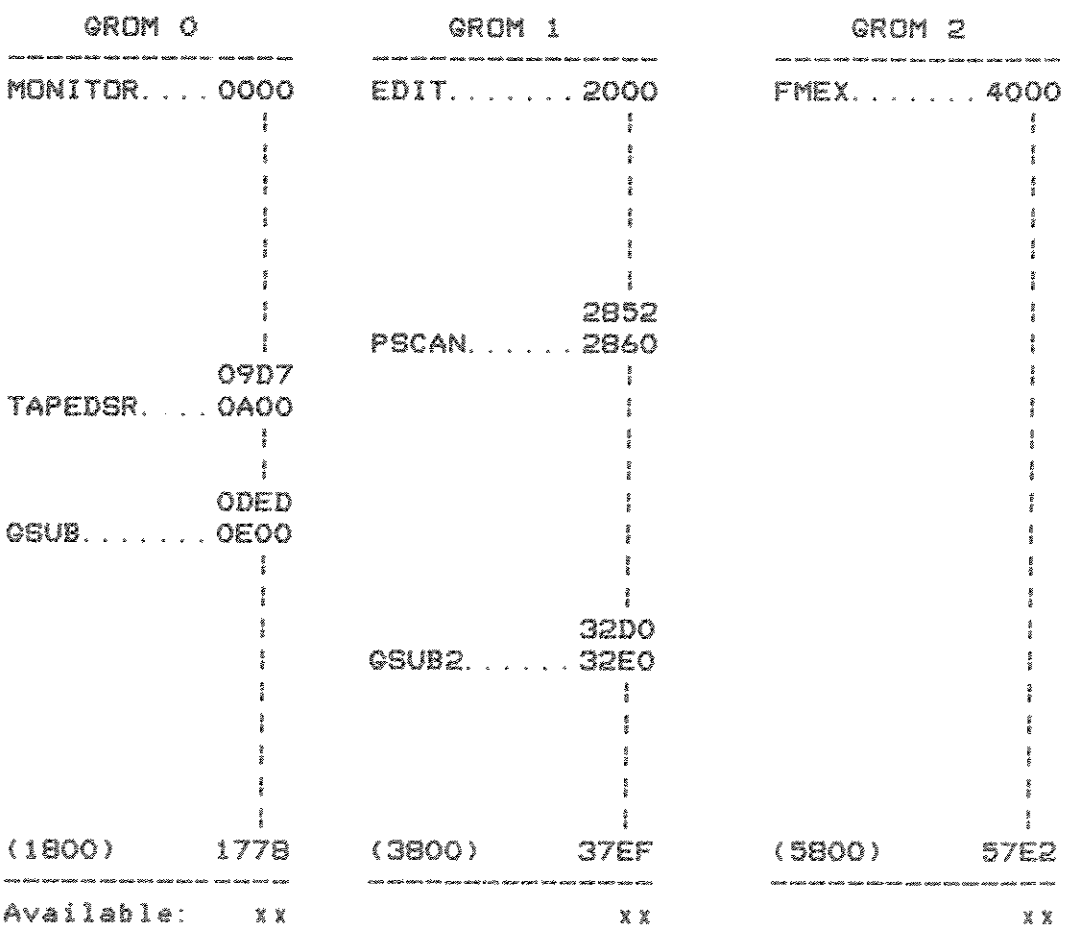


Figure 3-1 TI-99/8 GROM Memory Map

Each GROM holds 6K worth of information and has been fully utilized.

3.1.3 CPU RAM Usage.

The 2K bytes of on-board RAM are used exclusively for the Scratch Pad RAM. The Graphics Language interpreter, various interrupt routines, and peripheral devices use 142 bytes from >8372 to >83FF. The rest of the 2K bytes is used by the BASIC interpreter. This section itemizes BASIC's usage of the Scratch Pad RAM and generalizes the GPL interpreter's usage of the remaining bytes. Note that the GPL interpreter's workspace (>83E0 to >83FF) is also used by the assembly language portions of BASIC, so it is important to preserve information that the GPL interpreter needs. This information includes workspace registers 13, 14, and 15.

The following figure shows the layout of the Scratch Pad RAM using logical addresses.

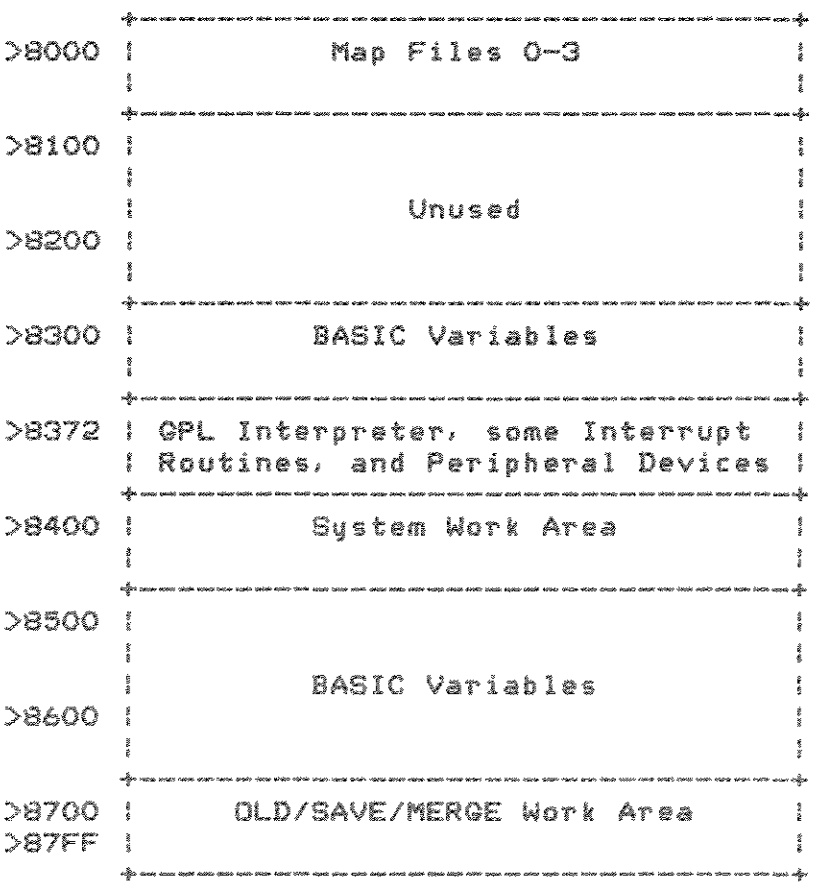


Figure 3-2 Scratch Pad RAM Description

3.1.4 VDP RAM Usage.

The VDP RAM is used as the primary memory for the screen, color table, character generator table, sprite information, and sound list. The current graphics mode determines how the VDP RAM is partitioned for use by the BASIC interpreter. More information on the data structures used in these areas can be found in the particular sections of this document which describe them. The following tables describe VDP RAM usage for each graphics mode.

Table 3-1 VDP RAM Usage for Pattern Mode

Addresses	Use
0000-02FF	Screen (768 bytes)
0300-037F	Sprite Attribute List (128 bytes)
03C0-03DF	Color Table (32 bytes)
03E0-03FF	Sound List (32 bytes)
0780-07FF	Sprite Velocity List (128 bytes)
0800-0FFF	Character Generator Table (2K bytes)
0800-0FFF	Sprite Generator Table (2K bytes)

Text mode usage of VDP RAM is similar to that of Pattern mode. The difference is that since the screen takes up more memory in Text mode, the Sprite Attribute List is at address >0400 instead of >0300.

Table 3-2 VDP RAM Usage for Text Mode

Addresses	Use
0000-03BF	Screen (960 bytes)
03C0-03DF	*Color Table (32 bytes)
03E0-03FF	Sound List (32 bytes)
0400-047F	*Sprite Attribute List (128 bytes)
0780-07FF	*Sprite Velocity List (128 bytes)
0800-0FFF	Character Generator Table (2K bytes)
0800-0FFF	*Sprite Generator Table (2K bytes)

* The sprite- and color-related tables are maintained even though sprites are not displayed in text mode. This is to insure that a CALL SPRITE command will have a safe place to write data.

The two Split-Screen modes and the High-Resolution mode, use completely different VDP RAM locations than the Pattern and Text modes.

Table 3-3 VDP RAM Usage for Split-Screen: Text-High Mode

Addresses	Use
0000-07FF	Sprite Generator Table (2K bytes)
0000-17FF	Character Generator Table (6K bytes)
1800-1AFF	Screen (768 bytes)
1B00-1B7F	Sprite Attribute List (128 bytes)
1B80-1BFF	Sprite Velocity List (128 bytes)
1C00-1C1F	Sound List (32 bytes)
2000-37FF	Color Table (6K bytes)

The two Split-Screen modes are the same except for the location of the Sprite Generator Table.

Table 3-4 VDP RAM Usage for Split-Screen: Text-Low Mode

Addresses	Use
0000-17FF	Character Generator Table (6K bytes)
1000-17FF	Sprite Generator Table (2K bytes)
1800-1AFF	Screen (768 bytes)
1B00-1B7F	Sprite Attribute List (128 bytes)
1B80-1BFF	Sprite Velocity List (128 bytes)
1C00-1C1F	Sound List (32 bytes)
2000-37FF	Color Table (6K bytes)

The High-Resolution mode has the Sprite Generator Table at >3800, which is different from both the Split-Screen modes. This mode also has an I/O Screen which is unique to the High-Resolution and Multicolor modes. The I/O Screen is the location where the results from the ACCEPT, DISPLAY, INPUT, LINPUT, and PRINT statements are placed. This causes the results from these statements to be invisible to the user.

Table 3-5 VDP RAM Usage for High-Resolution Mode

Addresses	Use
0000-17FF	Character Generator Table (6K bytes)
1800-1AFF	Screen (768 bytes)
1B00-1B7F	Sprite Attribute List (128 bytes)
1B80-1BFF	Sprite Velocity List (128 bytes)
1C00-1C1F	Sound List (32 bytes)
1D00-1FFF	I/O Screen (768 bytes)
2000-37FF	Color Table (6K bytes)
3800-3FFF	Sprite Generator Table (2K bytes)

The Multicolor mode is similar to Pattern mode, except the Sprite Generator Table is located at address >1000, rather than at >0800. This mode also has an I/O Screen which is unique to the High-Resolution and Multicolor modes. The I/O Screen is the location where the results from the ACCEPT, DISPLAY, INPUT, LINPUT, and PRINT statements are placed. This causes the results from these statements to be invisible to the user.

Table 3-6 VDP RAM Usage for Multicolor Mode

Addresses	Use
0000-02FF	Screen (768 bytes)
0300-037F	Sprite Attribute List (128 bytes)
03C0-03DF	*Color Table (32 bytes)
03E0-03FF	Sound List (32 bytes)
0780-07FF	Sprite Velocity List (128 bytes)
0800-0FFF	**Character Generator Table (2K bytes)
1000-17FF	Sprite Generator Table (2K bytes)
1D00-1FFF	I/O Screen (768 bytes)

* The Color Table is not useful in this mode, but it is defined.

** The Character Generator Table is actually used for color information in this mode.

3.1.5 Physical Memory Usage.

The following figure is a graphic representation of how physical memory is partitioned for use by the BASIC interpreter.

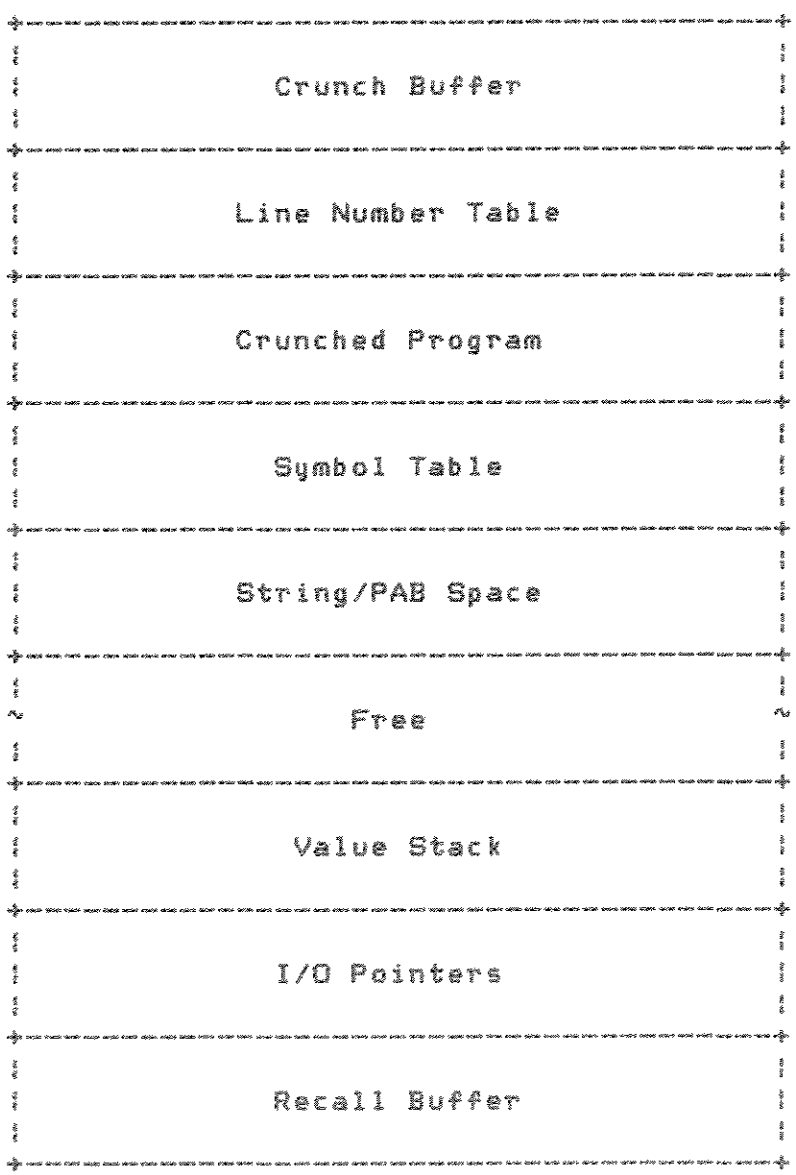


Figure 3-3 TI-99/8 Mapped Memory Usage

3.2 System Description

The TI-99/8 system has a 9995 processor with 64K of physical memory in the console and is expandable to 16 MEG of physical memory. There are two possible modes: /8 mode and /4A mode. Pascal uses the /8 mode and BASIC uses the /4A mode. This document only describes the /4A mode.

Since there are 16 address lines from the 9995 and 24 address lines to physical memory, there is a need for intermediary addressing logic. The Address Decoder and the Memory Mapper chip are the intermediary components for the TI-99/8. The following figure shows an overview of the system.

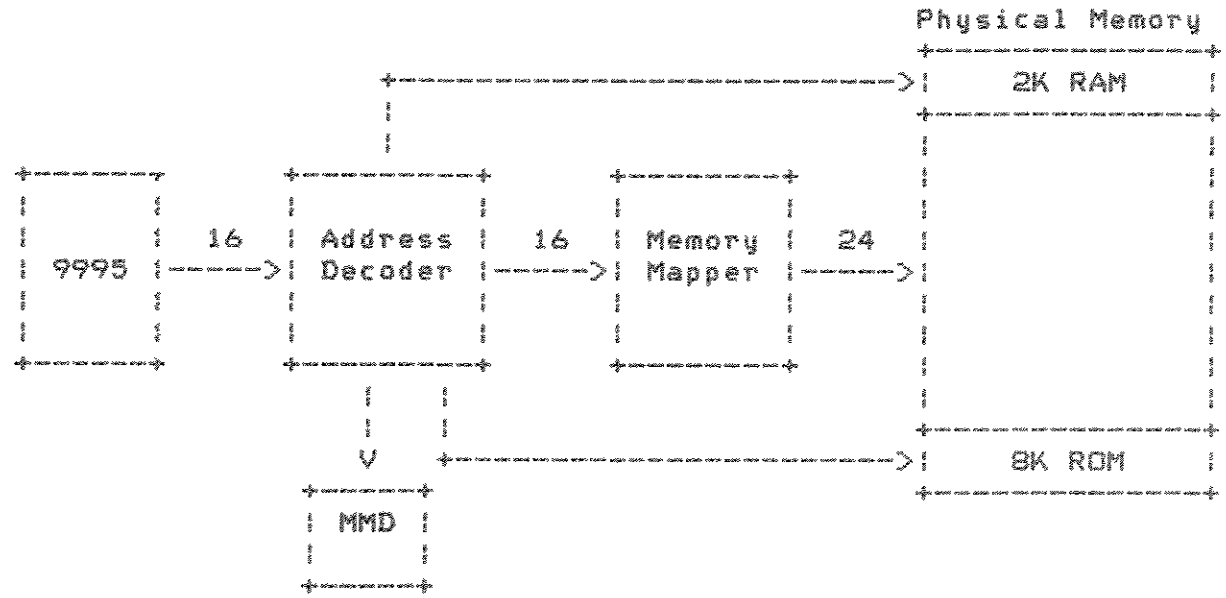


Figure 3-4 System Overview

Memory is generally described in two ways: Logical Address Space (LAS) and Physical Address Space (PAS). The LAS is merely an aid for the programmer and is a logical representation of how the PAS is being used. Valid logical addresses range from >0000 to >FFFF. The PAS is the actual memory which includes console memory and expanded memory. Valid physical addresses range from >000000 to >FFFFFF. The following figure describes the layout of physical memory.

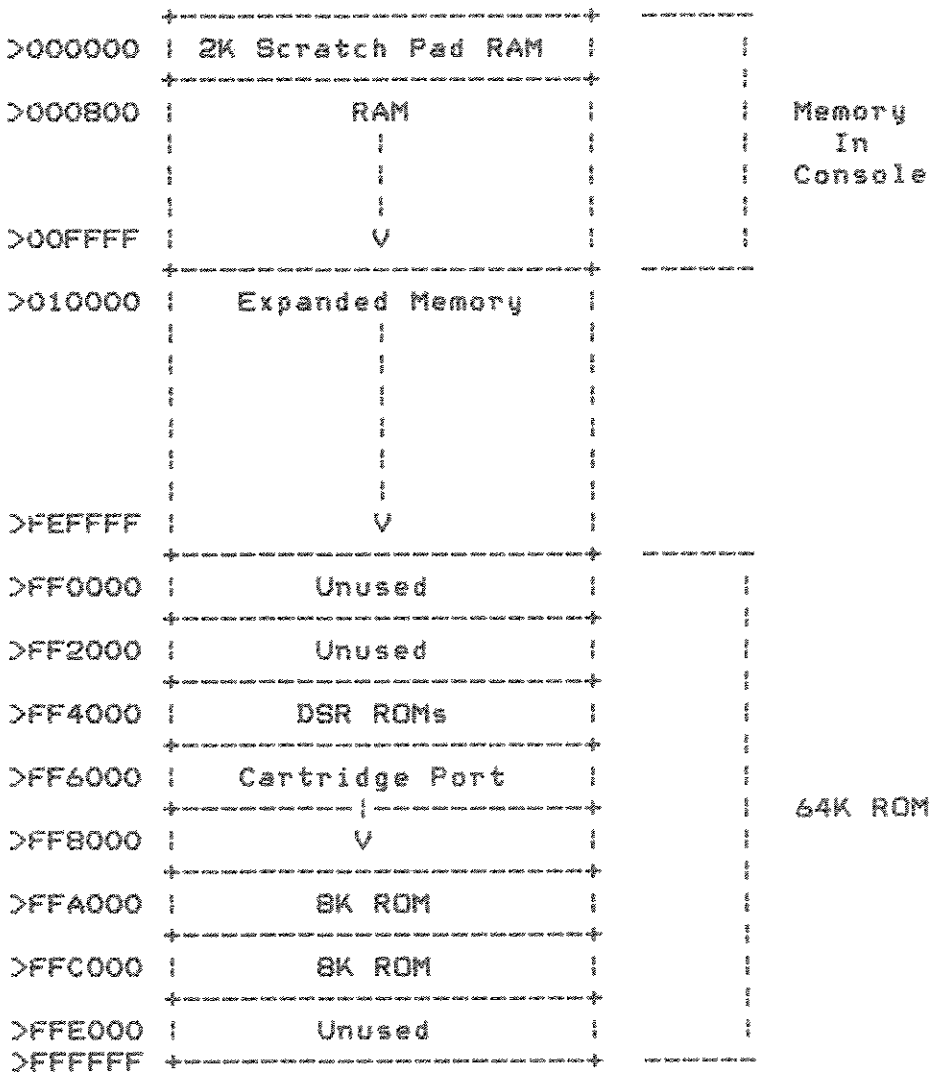


Figure 3-5 Physical Memory

3.2.1 The Address Decoder.

The Address Decoder takes logical addresses and routes them to the appropriate place. Logical addresses >0000 through >1FFF go to the system ROM. (The system ROM, or ROM 0, is not in the physical address space.) Logical addresses >8000 through >87FF are translated to physical address >000000 through >0007FF respectively, and sent to the 2K Scratch Pad RAM. Logical addresses >8800 through >9FFF are sent to the appropriate Memory

Mapped Device (MMD). All other logical addresses are sent to the Memory Mapper.

3.2.2 The Memory Mapper.

The Memory Mapper translates logical addresses from the Address Decoder into physical addresses. To do this, the Memory Mapper uses the value in the most significant nibble of the logical address as the address of one of its 16 registers. It then uses the value in the least significant 3 nibbles of the logical address as a displacement, and adds it to the contents of that register. The result is the desired physical address and is placed on the 24-bit address bus to physical memory.

3.2.3 Map Files.

In order to change the way PAS is partitioned, it is necessary to define a map file. A map file contains information which describes the 4K sections of PAS which are to be mapped into the LAS. A map file can only describe PAS sections which are 4K bytes in length.

There are eight map files numbered 0 to 7. Multiple map files may be defined, but only one map file at a time can be used to map in the PAS. The following table shows the logical address of each map file.

Table 3-7 Map File Locations

Map File	Logical Address
0	>8000
1	>8040
2	>8080
3	>80C0
4	>8100
5	>8140
6	>8180
7	>81C0

Each map file uses 64 bytes, and is divided into sixteen 2-word entries. Each entry of the current map file describes a "window" in the LAS. Information for each entry is stored as follows. The three most significant bits of the first byte are for the flags not used in the system. These three flags are the execute protect, the read protect, and the write protect flags. The remaining five bits of the first byte are unused. The

second, third, and fourth bytes (24 bits) store the physical address.

Only the first two map files, map file 0 and map file 1, are currently used by BASIC, but map file 2 and map file 3 are reserved for future use. The current definition does not reserve >8100 - >8200 for map files.

3.2.4 The Map Register.

Storing an appropriate value in the map register tells the Memory Mapper to LOAD or READ the current memory configuration described by the specified map file. This can be done as often as the programmer desires. The 8 bit map register is located at address >8810. The definition of each bit in the map register is described below.

U = Unused
A = Map File Number
L = 1 for LOAD a map file into the memory mapper
L = 0 for READ current memory mapper into a map file

Figure 3-6 Map Register Description

Specifying LOAD tells the memory mapper which 4K sections of PAS are to be mapped into the LAS. Specifying READ puts the sixteen 2-word physical address pointers of the currently mapped PAS into the specified map file.

The following table describes the results of storing various values in the map register.

Table 3-8 Map Register Contents

Map File	Map Register Contents			Meaning
0	0000 0000	or	>00	READ Map File 0
0	0000 0001	or	>01	LOAD Map File 0
1	0000 0010	or	>02	READ Map File 1
1	0000 0011	or	>03	LOAD Map File 1
2	0000 0100	or	>04	READ Map File 2
2	0000 0101	or	>05	LOAD Map File 2
3	0000 0110	or	>06	READ Map File 3
3	0000 0111	or	>07	LOAD Map File 3

Examples of how to use the Map Register are described later in this section.

3.2.5 Memory.

The Logical Address Space is the only memory which is directly addressable by the processor. The LAS contains 16 "windows". Each "window" corresponds to a logical address as follows: Window 0 is address >0000, Window 1 is address >1000, Window 2 is address >2000, etc. Each "window" contains a physical address which points to the top of a 4K section of PAS and is defined by the current map file. So if the current map file is Map File 0, the contents of Window 0 corresponds to information stored at logical address >8000, and the contents of Window 1 corresponds to information stored at logical address >8004, etc. Note that the contents of Window 0, Window 1, Window 8, and Window 9 can not be changed.

In order to access physical memory, it is necessary to do memory mapping. In other words, it is necessary to map physical addresses into logical address windows.

Map file 0 starts at Scratch Pad address >8000. Each of the sixteen 2-word entries in the map file contain information about the PAS and correspond to part of the LAS. Figure 3-7 shows the contents of Map File 0, the corresponding logical addresses, and the layout of physical memory at power-up time.

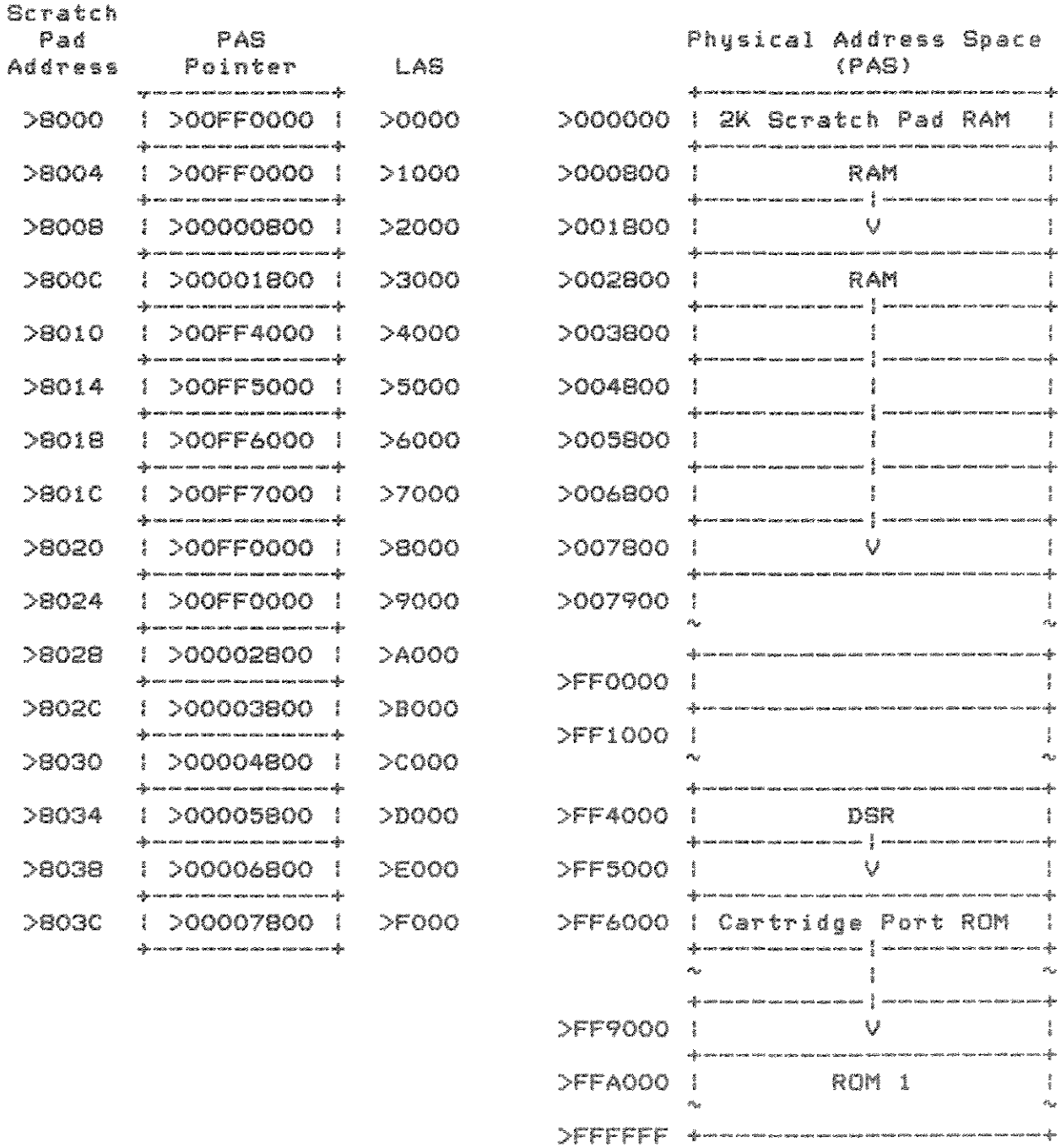


Figure 3-7 System Memory at Power-up Time

The following figure shows the contents of Map File 0, the corresponding logical addresses, and the layout of how the power-up code maps the PAS for BASIC.

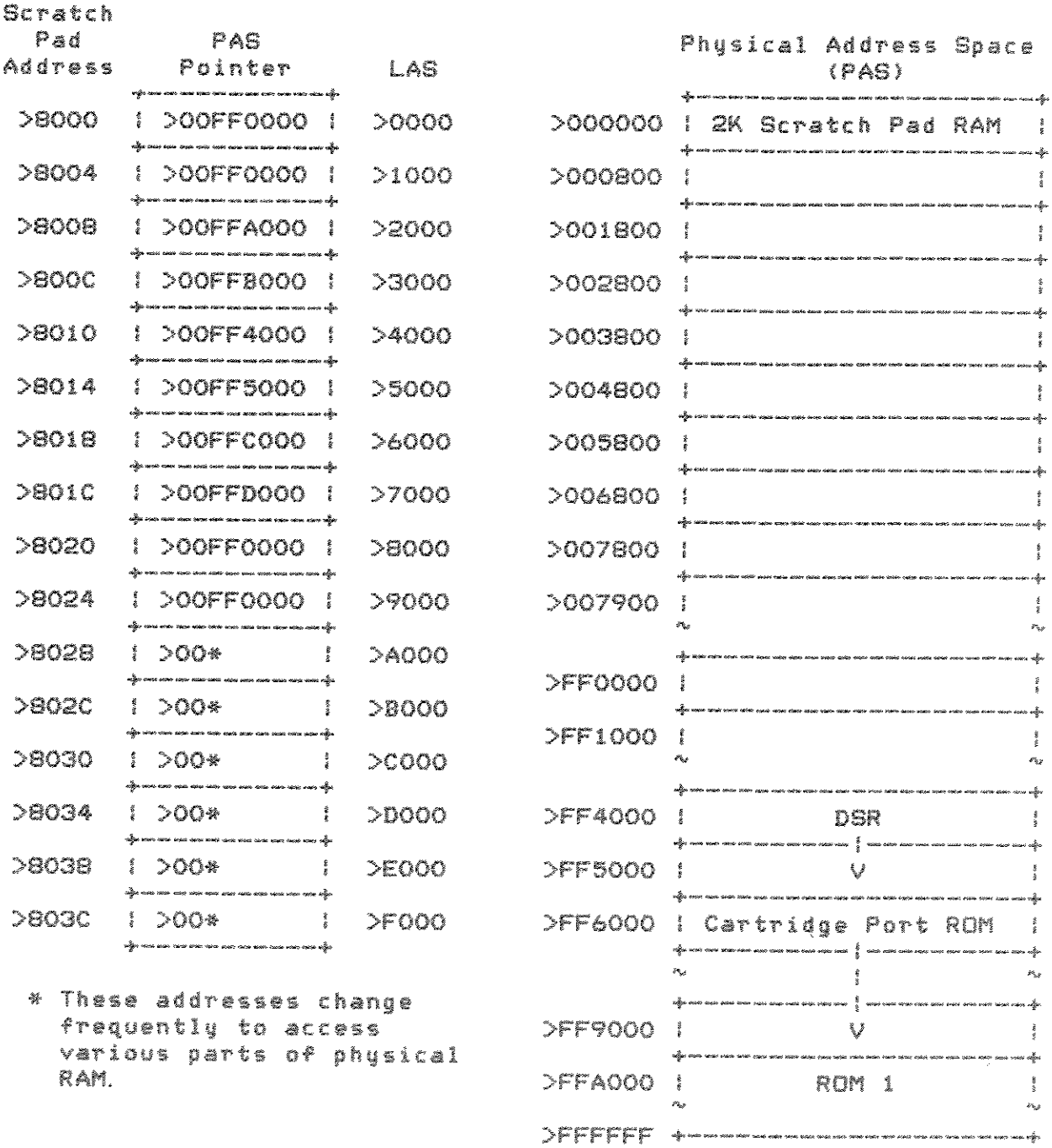


Figure 3-8 Memory Map of BASIC

The following paragraphs describe how to access the Memory Mapper from Assembly Language and from GPL.

3.2.6 Accessing the Memory Mapper From Assembly Language.

The following code assumes FAC through FAC+3 contain the 2-word physical address to be mapped in at window >D of map file 0.

```
WINDOW EQU  >8034              >8000+(4*>D)
LOADO  BYTE >01                00 (for MAP FILE 0) !! 1 (for LOAD)
MAPREG EQU  >8810              THE MAP REGISTER ADDRESS
*
       MOV  @FAC,@WINDOW       STORE 1ST HALF OF 2-WORD PHYSICAL
*                              ADDRESS TO MAP IN MAP FILE
       MOV  @FAC+2,@WINDOW+2    STORE 2ND HALF OF 2-WORD PHYSICAL
*                               ADDRESS TO MAP IN MAP FILE
       MOVB @LOADO,@MAPREG     SEND COMMAND TO LOAD MAP FILE 0
```

The two MOV instructions need to be repeated for each physical address to be mapped in. The MOVB instruction only needs to be executed once, since it will load (or read) the entire specified map file

The following XOP 3 instruction accomplishes the same procedure. It is slower than directly accessing the mapper because it is a subroutine call, but it saves program steps. For the example as set up above:

```
WINDOW EQU  >8034       >8000+(4*>D)

       XOP  @FAC,3      FAC CONTAINS A 2-WORD ADDRESS TO MAP IN
       DATA WINDOW      PUT MAP FILE ADDRESS IN DATA STATEMENT
```

The XOP 3 instruction needs to be repeated for each physical address to be mapped in. The XOP 3 instruction automatically loads the entire map file 0, so this instruction is very slow when several physical addresses need to be mapped in.

It is not necessary to define every window with MOV instructions. The following example shows how to read the current description of physical memory into map file 2. It is then possible to change any windows and then to load the new map file.

```
READ2  BYTE >04                10 (for MAP FILE 2) !! 0 (for READ)
LOAD2  BYTE >05                10 (for MAP FILE 2) !! 1 (for LOAD)
MAPREG EQU  >8810              THE MAP REGISTER ADDRESS

       MOVB @READ2,@MAPREG     SEND COMMAND TO READ CURRENT PHYSICAL
```

```
*                                   MEMORY POINTERS INTO MAP FILE 2
*       Change any windows.
*
        MOVB @LOAD2,@MAPREG     SEND COMMAND TO LOAD MAP FILE 2
```

3.2.7 Accessing the Memory Mapper From GPL.

The following paragraphs describe GPL instructions which are used in memory mapping.

3.2.7.1 MAPIN.

The MAPIN routine fetches the physical address pointer from a Scratch PAD RAM address (>8300 - >87FF), and maps this address in at the top of the 4K logical address space window specified by the WINDOW parameter. The format for this instruction is:

```
DATA  XML2,MAPIN
DATA  #PHYADD,WINDOW
```

In MAPIN, PHYADD (Physical Address) is a 1-word pointer to the Scratch Pad RAM location containing a 2-word physical address. WINDOW is 1-byte parameter and contains the value of the desired window. (Recall that windows have values from >0 through >F.)

An example for this instruction is:

```
DATA  XML2,MAPIN
DATA  #FAC+4,>E
```

This example moves the physical address at FAC+4 through FAC+7 to Window E of map file 0.

3.2.7.2 GETP.

The GETP routine will fetch data from a physical address to a buffer in Scratch Pad RAM (>8300 - >87FF). The format for this instruction is:

```
DATA  XML2,GETP
DATA  #SRCPTR,#LOGDEST,BYTCNT
```

In GETP, SRCPTR (Source Pointer) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer to data located in physical memory. LOGDEST (Logical Destination) is a 1-word address of where in Scratch Pad RAM to put the data. BYTCNT is a 1-byte parameter which tells how many bytes of data to read.

An example for this instruction is:

```
DATA  XML2,GETP
DATA  #DATA,#CHAT,1
```

This example moves 1 byte from the physical address pointed to by DATA to CHAT.

3.2.7.3 PUTP.

The PUTP routine will fetch data from a Scratch Pad RAM address to a physical address. The format for this instruction is:

```
DATA  XML2,PUTP
DATA  #SRCPTR,#PHYDEST,BYTCNT
```

In PUTP, SRCPTR (Source Pointer) is a 1-word address of where in Scratch Pad RAM the data is located. PHYDEST (Physical Destination) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer of where in physical memory to put the data. BYTCNT is a 1-byte parameter which tells how many bytes of data to read.

An example for this instruction is:

```
DATA  XML2,PUTP
DATA  #FAC,#ARG,4
```

This example moves 4 bytes from FAC to the physical address pointed to by ARG through ARG+4.

3.2.7.4 MOVUP.

The MOVUP routine will move the contents of low physical memory to high physical memory. The format for this instruction is:

```
DATA  XML2,MOVUP
DATA  #SRCPTR,#PHYDEST,#LENGTH
```

In MOVUP, SRCPTR (Source Pointer) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer to the data located in physical memory. PHYDEST (Physical Destination) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer of the address of where in physical memory to put the data. Since MOVUP is a non-destructive move, it is necessary for SRCPTR to point to the high-end address of the buffer that is to be moved. Similarly, PHYDEST must point to the high-end address of the desired buffer location. LENGTH is a 1-word address of a length

variable.

An example for this instruction is:

```
DATA  XML2,MOVUP
DATA  #OLDBOTTM,#NEWBOTTM,#FAC
```

This example moves FAC number of bytes from the physical address pointed to by OLDBOTTM to the physical address pointed to by NEWBOTTM.

3.2.7.5 MOVDN.

The MOVDN routine will move the contents of high physical memory to low physical memory. The format for this instruction is:

```
DATA  XML2,MOVDN
DATA  #SRCPTR,#PHYDEST,#LENGTH
```

In MOVDN, SRCPTR (Source Pointer) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer to the data located in physical memory. PHYDEST (Physical Destination) is a 1-word address in Scratch Pad RAM which contains a 2-word pointer of the address of where in physical memory to put the data. Since MOVDN is a non-destructive move, it is necessary for SRCPTR to point to the low-end address of the buffer that is to be moved. Similarly, PHYDEST must point to the low-end address of the desired buffer location. LENGTH is a 1-word address of a length variable.

An example for this instruction is:

```
DATA  XML2,MOVDN
DATA  #OLDTOP,#NEWTOP,#FAC
```

This example moves FAC number of bytes from the physical address pointed to by OLDTOP to the physical address pointed to by NEWTOP.

SECTION 4

INTERPRETER PHASES

This section describes the different phases involved in executing a BASIC statement or program. There are three major phases involved in executing a BASIC statement or program. These phases are:

1. Statement/program entry (editing),

2. Statement/program prescanning, and

3. Statement/program execution.

Each of these phases have several subphases which 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 (a 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, or conditionally with an up-arrow or a down-arrow depending upon the particular mode which called the routine.

READLN has three modes of operation when used for inputting 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 when 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 inputting 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 reentering 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 exited 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. The READLN routine 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 read routine. 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 FUNC-S as a back-arrow and moves the cursor back one position on the screen, wrapping around to the previous line if at the beginning of a continuation line in the current line. Similarly, the line editor recognizes the FUNC-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 line until 255 characters have been entered.

The line editor also recognizes two other keys as being special inline editing keys. The FUNC-1 is recognized as the delete character key and deletes whatever character is located at the cursor's position and shifts 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 FUNC-2 key is recognized as putting the line editor into insert mode. After a FUNC-2 key is recognized all characters subsequently recognized are inserted into the line in the position immediately preceeding the cursor. Any characters to the right of the cursor which meet the end of the input buffer are bumped off. Insertion is no longer possible when the cursor reaches the end of the input buffer.

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 TI-99/8 BASIC interpreter uses a "crunched" version of the actual BASIC program for direct execution. This enhances the speed with which each program line can be executed since all lines are already preprocessed.

This preprocessing 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 and CALL 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 preprocessed 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 maximum length of a "crunched" line is 160 characters.

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 Program Line Representation

The line number of each program line is stored in a separate "line number table". This table is physically located below the program text segment in VDP 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 controlled by three pointers:

1. ENLN (CPU >8302) - Indicates last location used by the line number table. This pointer is pointed at the least significant byte of the program line pointer.

2. STLN (CPU >8300) - 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.

3. PGMTOP (CPU >8304) - Points to the first byte past the end of the program text.

Note: MEMTOP (CPU >8370), is maintained as VDP top-of-memory, but is not used by TI Extended BASIC II.

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.

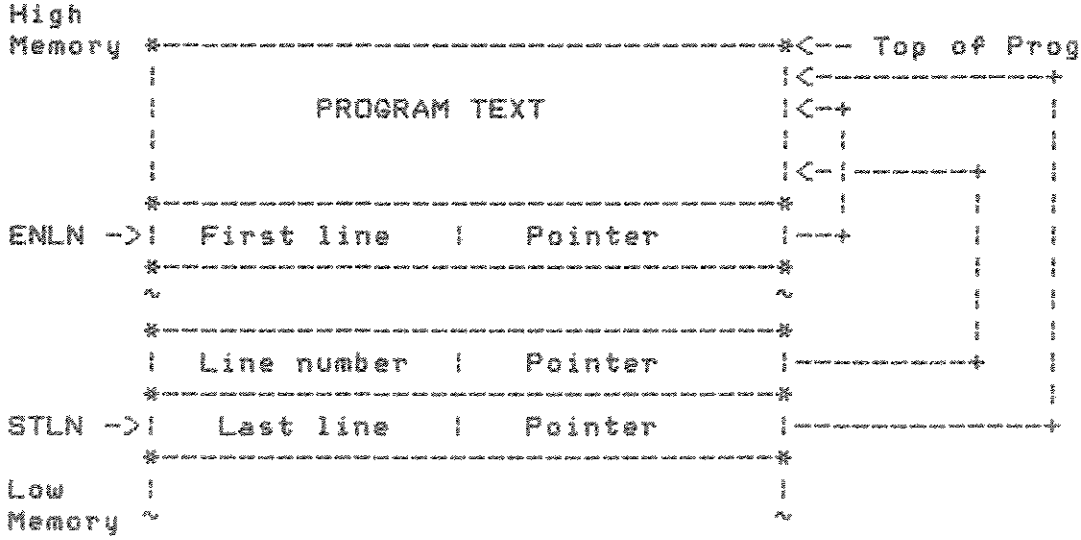


Figure 4-2 Program Memory Image

The assignment of a breakpoint causes the most significant bit of the line number to be set in the line number table.
Note: The pointers are kept as physical pointers, with the most significant word implied to be zero.

4.1.3.1 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.

4.1.4 Auto-Num.

The auto-num feature is included in the TI-99/8 BASIC to provide for ease in entering programs into the machine. 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 TI-99/8 BASIC interpreter. The list command has the ability to list a portion of a program or an entire program, to either the display or a device. Execution of the list command begins by searching 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. 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 MAIN1 which initializes the symbol table and string space.

4.1.6 Resequence.

Execution of the resequence feature of the TI-99/8 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. 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, and that INPUT, DATA, FOR, NEXT, GOTO, GOSUB, DEF, RETURN, etc. statements are not used imperatively. 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 4-3 below.

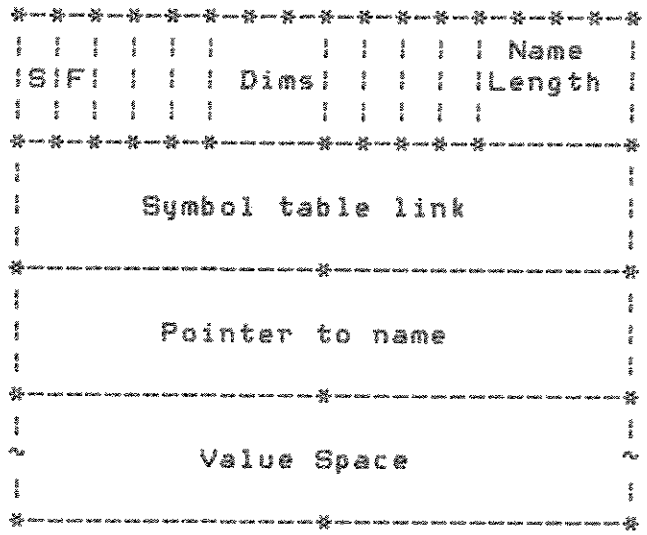


Figure 4-3 Symbol Table Entry

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
Dims - Number of dimensions or function parameters - Legal values 0 - 3
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 entry(ies) for strings;
* 2-byte pointer to definition for User-Defined functions.

4.3 Execution

A BASIC program or imperative statement is executed by the use of the EXEC command in GPL. When the EXEC 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 variable, DATA, 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

DATA is moved to EXTRAM to allow the program to begin execution at some line other than the first one. The routine DATAST is called to set up the data pointer for READ and DATA statements. 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 GPL EXEC command is used to execute one or more lines of BASIC. The op code for EXEC is found in one of the GPL interpreter's tables 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 - >A2) of the valid statement tokens. 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 the code to handle it 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 top bit is 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, a GPL PARSE command cannot occur in any GROMs except GROMs 0 through 3. Fot this reason, any GPL subprograms called from BASIC which need to parse a value must reside in GROM 3.

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 and 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 in a program or to return to top-level.

4.3.2 PARSE.

The parsing algorithm used in the TI-99/8 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 '&', '=', '<', '>', '+', '-', '*', 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 CONT command and is accessed from assembly language by the execution of a

```
B  @CONT
```

instruction.

SECTION 5

INTERPRETER COMPONENTS

5.1 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.

5.1.1 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 and GOSUB statements and references to User-Defined statements 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 entries are placed 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 and 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 >600 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:

Address of String Pointer	>65		Address Of String	Length of String

Figure 5-1 Stack/FAC Entry for a String

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.

2. FAC+2 - >65 is the string identification byte

3. FAC+3 - Unused

4. FAC+4,FAC+5 - Address of the first character of the string

5. 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:

Return line Table Pointer	>66			

Figure 5-2 Stack Entry for a GOSUB Statement

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

The stack entry for a FOR statement looks like:

Ptr. to Symbol Table Entry	>67		Value Space Pointer	Old Line Number Table Pointer
Increment Value				
Loop Limit				

Figure 5-3 Stack Entry for a FOR Statement

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 - Line number table pointer to FOR line

Entry 2
FAC-FAC+7 - Value of increment for index variable

Entry 3
FAC-FAC+7 - Value of index limit

The stack entry for a User-Defined function looks like:

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

Figure 5-4 Stack Entry for a User-Defined Function

1. FAC,FAC+1 - Return line pointer (PGMPTR)
2. FAC+2 - >68 is the User-Defined function identification byte.
3. FAC+3 - Function type, >00=numeric, >80=string
4. FAC+4,FAC+5 - Return symbol table pointer (SYMTAB)
5. FAC+6,FAC+7 - Return free space pointer (SYMPTR)

5.1.2 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 garage collection must be performed.

User memory in the Home Computer between addresses >600 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:

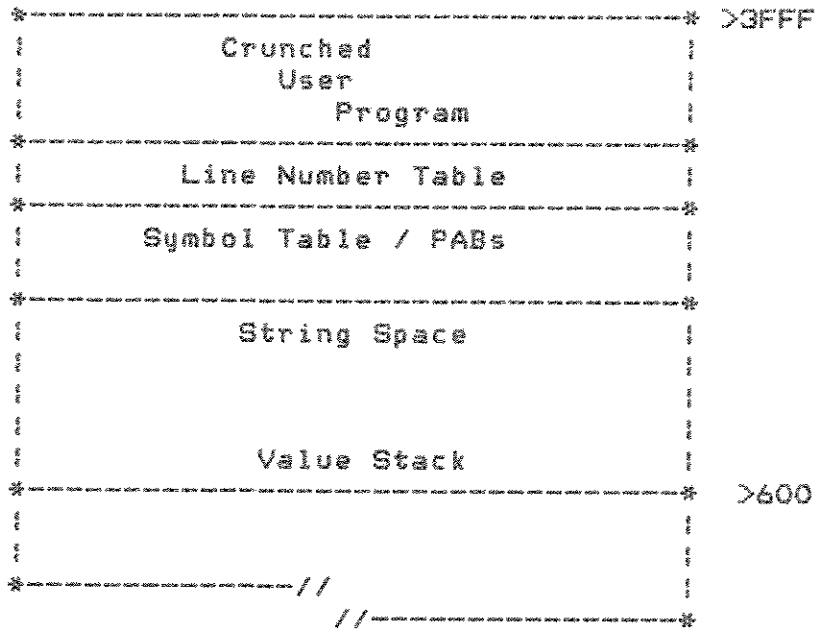


Figure 5-5 Memory Usage

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 arise 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.

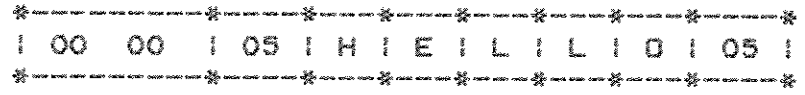
The symbol table and PABs present a problem in that there can be, and probably are, strings in the string space when a symbol table or PAB entry is allocated. Symbol table entries and PAB's occupy contiguous memory and when an entry is allocated the entire string space is moved down (lower address) in 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).

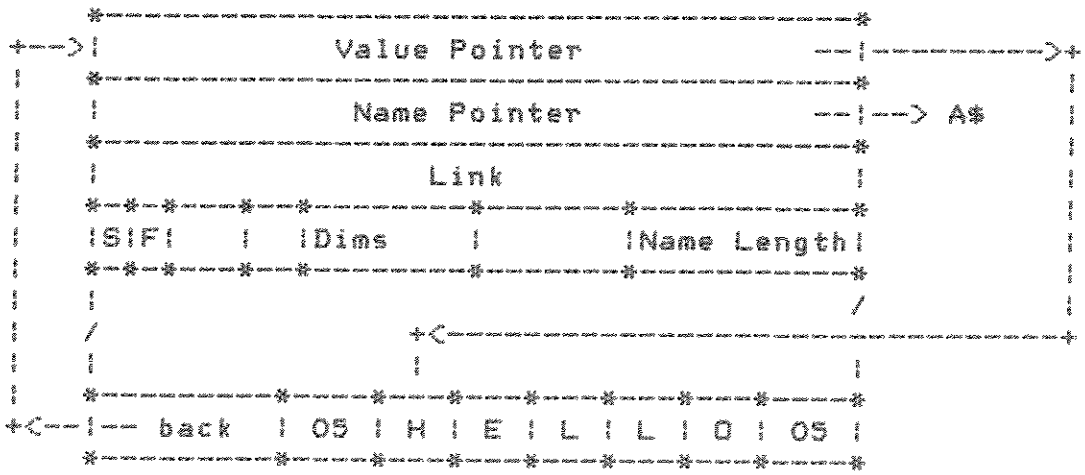
Following is a detailed description of how strings are created, used, and destroyed in the course of executing a BASIC program.

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.



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.

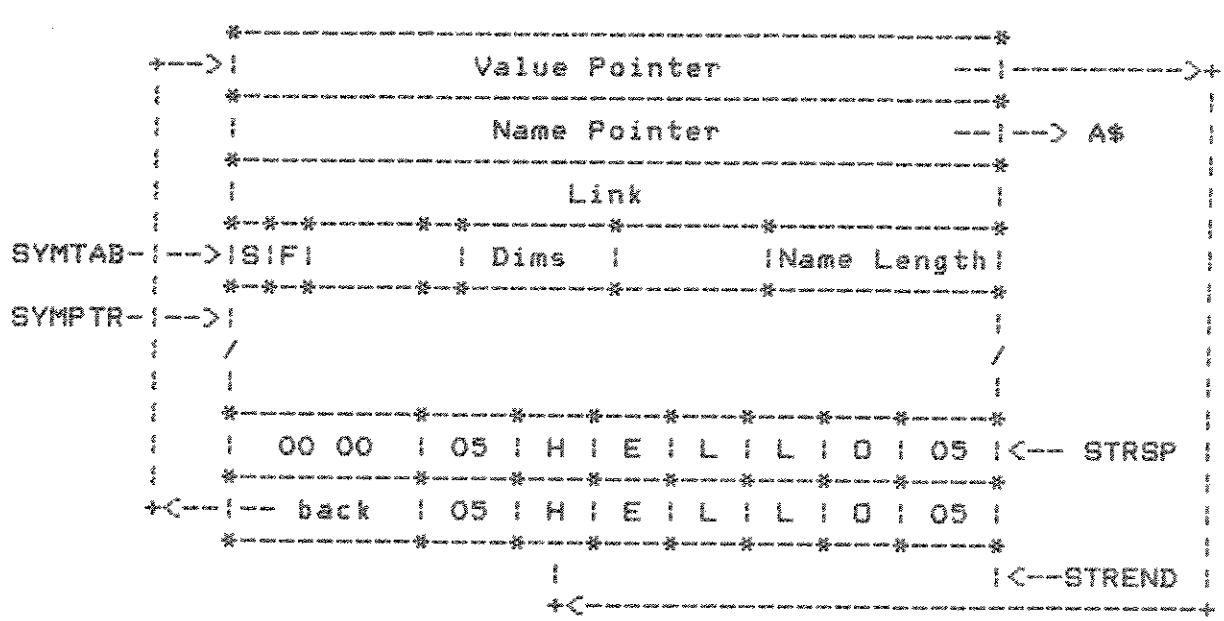
If the string is assigned to a variable, then the following is a graphic representation of the final result.



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.

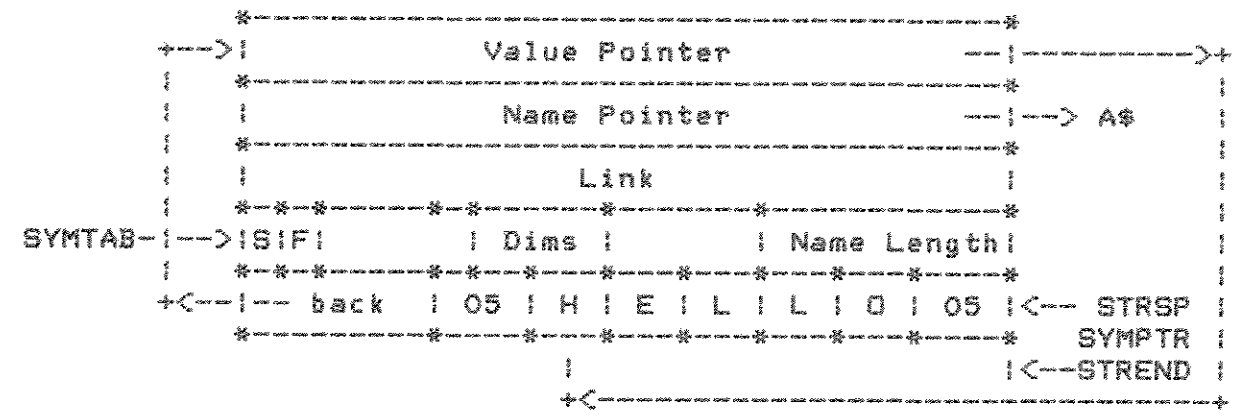
When, for example, a PRINT A$ statement is executed the pointer is picked up from the symbol table, one is subtracted from it to get the length, 5, and the next five bytes

(characters) are printed. If an A$=A$ statement is executed, a completely new string is created, the old one is freed and the new one is assigned to A$. Using this example, memory would now look like:



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 (nonzero backpointer), it is moved up in memory so that STRSP and SYMPTR 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:



Another important facet of the string scheme is the FAC entry which is created by the use of a string. The next two figures describe the FAC entries for temporary and permanent strings.

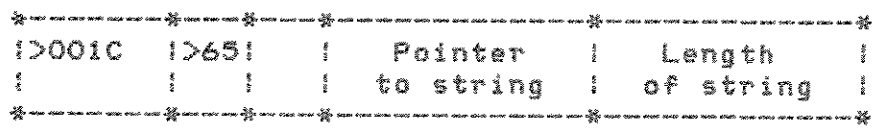


Figure 5-6 FAC Entry for a Temporary String

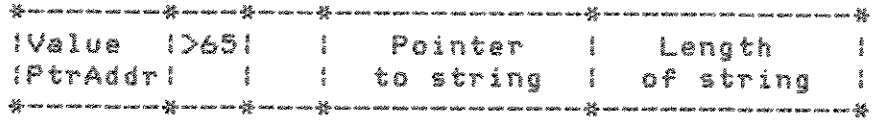


Figure 5-7 FAC Entry for a Permanent String

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 relieve all other codes from the responsibility of getting the value pointer back again.

5.2 Math Package

The Math package used in the TI-99/8 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.

The format used in this implementation is the following:

Figure 5-8 Real-Type Numeric Data Format

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.

5.3 String Package

The string package of the TI-99/8 BASIC 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. 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:

1. 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 GROM so that it may be copied from the correct memory.

2. 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 <u>not</u> been copied into the string space. This is the responsibility of the caller after the string has been allocated.

5.4 Functions and Operators

This section describes how all of the functions and operators contained in the TI-99/8 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.

5.4.1 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 a CONT instruction is executed, otherwise an error message is issued.

Execution of all of the binary (LED) operators are very similar and common code is utilized to handle the common portions of each operator except involution (^) which is handled differently because it is handled in GPL and the rest of the operators are handled in assembly language. Each of the operators does a parse to a particular level.

Each of the operators, plus, minus, multiply and divide, 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 message can be issued in the case of an overflow, and then a branch to CONT is made to return control to the parser.

Involution (^) is handled specially in GPL because the involution routine contained in the system is written in GPL. The involution code in BASIC does exactly the same things as the assembly language code for the other operators. It insures that each operand is a numeric, calls the involution routine and then

issues a warning if an overflow occurs and then executes a GPL CONT instruction.

5.4.2 Arithmetic Functions.

Included in the TI-99/8 BASIC interpreter are twelve 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 ramaining functions include the absolute value (ABS), End Of File (EOF), exponential (EXP), integer (INT), 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 GPL and are contained in the EXEC assembly except for the EOF function which is contained in the FLMGR assembly of BASIC.

5.4.2.1 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 the MONITOR) into ARG 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 instruction following the PARSE instruction in COMMON (CA1) onto the top of the subroutine stack. The address of the function handler is then placed on to the top of the subroutine stack so that the PARSE instruction following it will return to the correct function evaluator in the MONITOR, evaluate the function, and then return to the label, CA1, in COMMON, completing the evaluation of the function. Common then checks for any errors or warnings, handles them apropriately and, if it is able to continue, executes a GPL 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. These algorithms will not be described in detail here, as they are not an integral part of BASIC, but are part of the systems' software contained in the TI-99/8.

5.4.2.2 Other Arithmetic Functions.

Of the remaining arithmetic functions included in the TI-99/8 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 of the MONITOR 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 monitor. The GRINT routine itself will not be described here, since it is not an integral part of BASIC but is contained in the systems' software.

The random number function (RND) generates the next number in a pseudo-random number sequence. It does this by first placing the exponent (>3F) of the radix-100 number to be generated into the FAC. It then uses the GPL RAND instruction to generate the next seven bytes of the number and places them into FAC+1 through FAC+7. Note that in order for a zero to be generated for the RND function, sixty-three zeroes in a row would have to be generated by the RAND instruction, which is known to be impossible (a look at the code will show why). After the RND function has built the FAC entry, a GPL CONT instruction is executed to return control to the parser.

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, a GPL 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, a GPL CONT instruction is executed to return control to the parser.

5.4.3 String Operators.

The only string operator supported by the TI-99/8 BASIC is that of concatenation (&). The concatenation operator is used to concatenate two strings together into one string.

5.4.3.1 Concatenation.

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

Getting to the LED handler for concatenation is not handled directly by the tables of the parser. In order for the concatenation operator to be added to the TI-99/8 BASIC, it was necessary to add a check in CONT for the operator and if it is present to send control to the LED handler. To add the concatenation operator to the LED table in PARSE would have cost several bytes and it was decided to add the detection of it directly to CONT.

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. 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 GPL CONT instruction.

5.4.4 String Functions.

This section will deal with the string functions ASCII (ASC), character (CHR$), length (LEN), position (POS), segment (SEG$), 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 GPL 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 a GPL 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 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, a GPL CONT instruction is executed to return control to the parser.

The string function (STR$) is used to convert a number into its string equivalent. It does this by first parsing the argument and converting it into a string by using the Convert Number to String (CNS) routine contained in the monitor. STR$ 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 a GPL 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 a GPL CONT instruction is executed to return control to the parser.

5.4.5 User-Defined Functions.

Execution of User-Defined functions is one of the more complicated parts of the TI-99/8 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 not added 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 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, execept 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 a GPL CONT instruction is executed to return control to the parser.

5.4.6 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.

5.5 Error Handling

A significant portion of the TI-99/8 BASIC interpreter is devoted to the detection and reporting 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 two sections describe what actions are taken by the interpreter when an error is detected.

5.5.1 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 two data bytes containing the address of the error message to be printed by the error routine. A sample calling sequence is:

```
MSGSNM DATA 22,:STRING-NUMBER MISMATCH:
          :
          :
       CALL ERR$$              Issue the error message
       DATA #MSGSNM            *STRING-NUMBER MISMATCH
```

The error message is STRING-NUMBER MISMATCH and its length is 22, which preceeds the message in the data-statement. When control reaches the error routine, it does two GPL FETCH instructions to get the address of the error message.

The preceeding discussion assumes that the error has been 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 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 message address following the call statement.

5.5.2 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 two bytes of the message address so it will know what error message to display on the screen. 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 scrolls the screen properly (all lines if in BASIC, bottom six lines if in Equation Calculator) and then displays an asterisk, which will preceed the error message on the screen. A subroutine, ERR$1, is called which is used by both the error routine and the warning routine, to display the error message on the screen and to display the line number that the error occurred in if a program was executing.

When control returns to ERR$$ after the error message has been displayed on the screen, a check is made to see if a program was being executed. If a program was being executed then the routine to close all open files (CLSALL) is called. After any open files have been closed, then the GROM flag is checked to return control to a GROM program if one was being executed. If a VDP RAM BASIC program was being executed the value stack is cleaned up by popping all temporary entries (string entries are critical here) and delinking a User-Defined function's parameter if a User-Defined function was being executed when the error occurred. The Equation Calculator flag is checked and if it is set control returns to the Equation Calculator. One final check is made of the program flag and if a program was being executed when the error was detected, the value stack is wiped out. Note that this is OK since all temporary entries have already been cleaned up. Finally, control returns to top-level by executing a BR MAO instruction to return to the editor.

5.5.3 Warnings.

Warning messages issued by the interpreter are handled in much the same way as error messages, except that execution continues after the message has been displayed. The only warning messages which can occur when a program is executing are a NUMERIC OVERFLOW message when a floating point operation has

taken place and the machine's capacity has been exceeded, and an INPUT ERROR message 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. The floating point routines all return an error code to tell if an overflow has occurred and the BASIC interpreter checks the error code to see if an an overflow has occurred. The input routine also detects its warnings and reports them.

The warning routine, WARN$$, is called in the same manner as the error routine, ERR$$, with the address of the message to be printed following the call statement, as in the following example:

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

The warning routine first scrolls the screen, if not in Calculator mode and then displays the message, '* WARNING:', on the screen. It then fetches the address of the message and scrolls the screen again. Control then flows into the common routine ERR$1 which displays the message and returns to the caller of the warning routine to resume execution of the violating statement in the normal manner.

SECTION 6

BASIC STATEMENTS AND SUBPROGRAMS

6.1 Introduction

This section describes, in detail, how each of the different statements in TI-99/8 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.

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. SMB will evaluate the subscripts in the event that the variable is an array. SMB places the address of where the value is to go into FAC+4. An identification byte of either >00 or >65 is added in FAC+2 to indicate to ASSGNV whether the symbol is a numeric or string, respectively. The 8-byte FAC entry, after SYM and SMB have been called looks like:

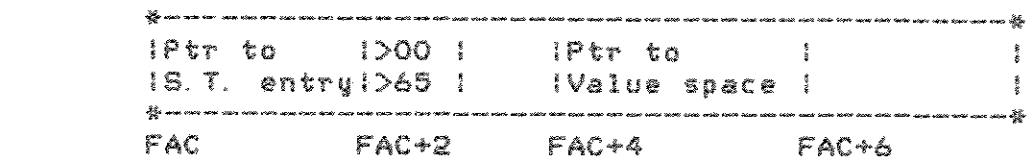


Figure 6-1 8-Byte FAC Entry

After this 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.

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.

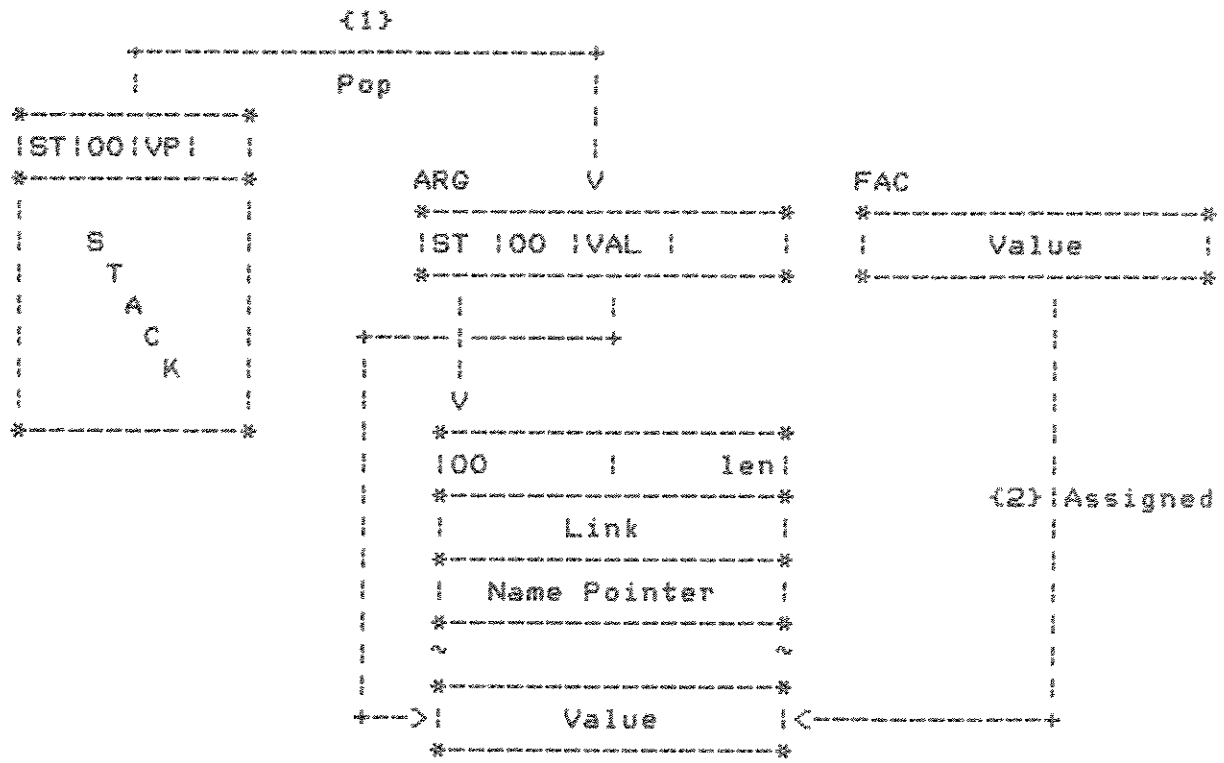


Figure 6-2

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

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.

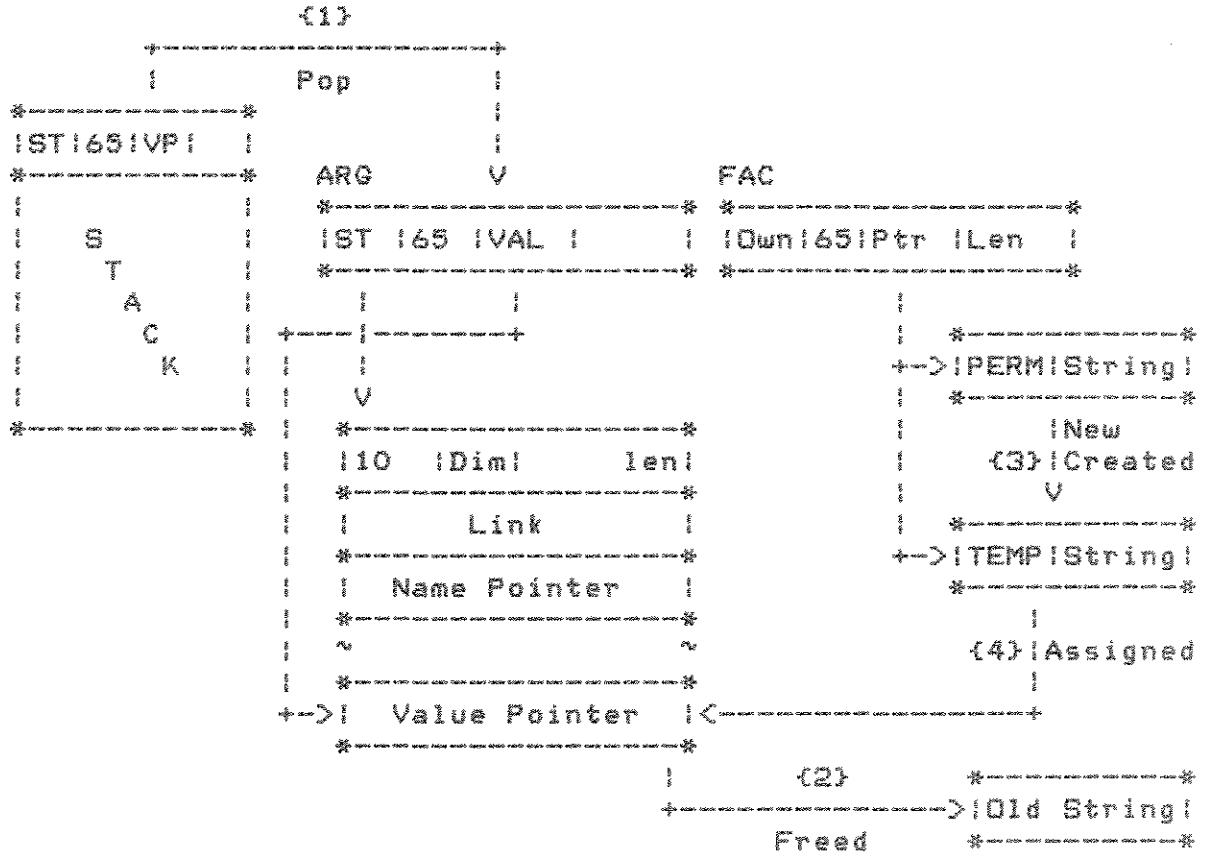


Figure 6-3

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.

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. 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. A check is made to see if an equal sign is present and if it is, the value to be assigned to the variable is parsed.

Next, the value parsed is assigned to the variable by using the ASSGNV routine described in the preceeding two sections.

6.3 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.

6.3.1 Screen and Keyboard.

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

6.3.1.1 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 initialize 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. Note that this is at the entry point for the DISPLAY statement.

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-line 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-line is reached. When the end-of-line is reached, a GPL '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.

```
PRINT     IF DEVICE-OUTPUT THEN
            INITIALIZE_PAB
          ELSE IF SCREEN
DISPLAY     INITIALIZE_SCREEN
          END IF
          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
```

6.3.1.2 DISPLAY Statement.

The DISPLAY statement is executed in exactly the same manner as a PRINT statement to the screen. Execution of the DISPLAY statement actually begins at the place where the PRINT statement calls INITKB to initialize I/O to the screen. A complete explaination of how the DISPLAY statement is executed may be found in the preceeding section on the PRINT statement.

6.3.1.3 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 incoming 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, a GPL CONT instruction is executed to return to the parser.

6.3.2 Device/File.

An extensive device/file I/O system is supported by the TI-99/8 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.

6.3.2.1 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. 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 6-4 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.

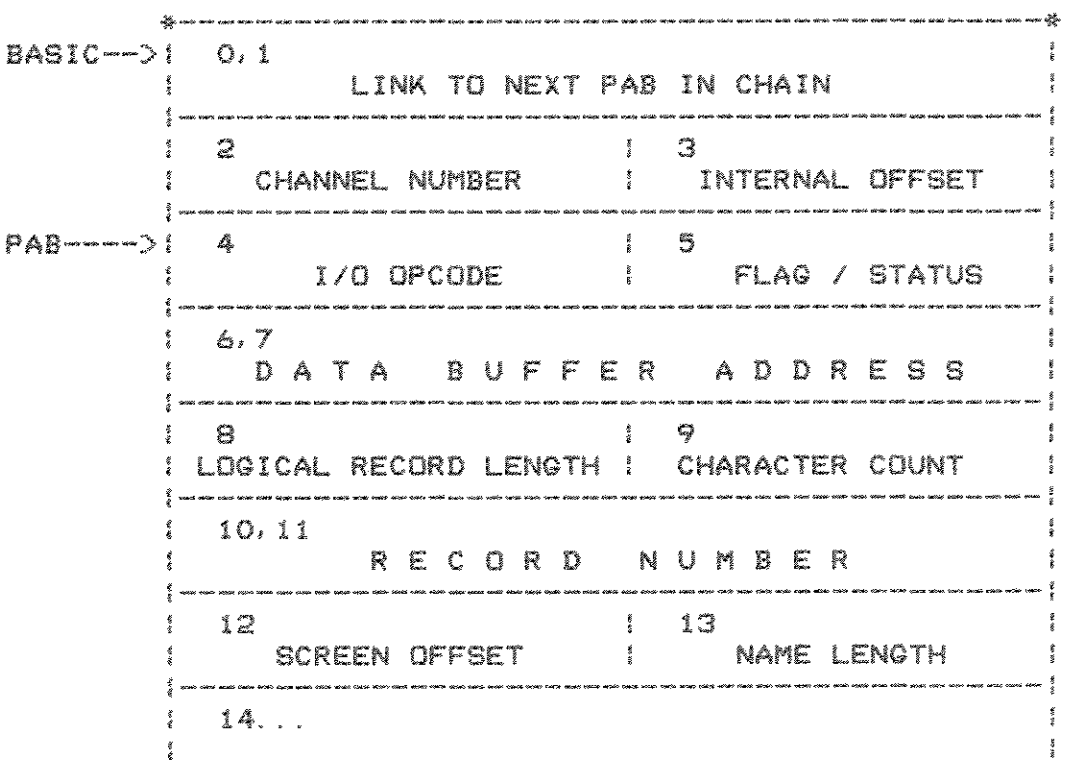


Figure 6-4 BASIC PAB Layout

The meaning of the bytes within the PAB is 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	-	Flag/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 in VDP RAM 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 - The device name and, if required, the filename and options. The length of this descriptor is given in byte 9.

6.3.2.2 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.

6.3.2.3 I/O Operations.

The valid opcodes which may appear in a PAB are shown in Figure 6-5. 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 6-5 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 verified 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 a GPL 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 opcode or the DELETE opcode (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, an End Of File (EOF) 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 or file. All of the control information the application program needs should be concatenated to the program image. No intermediary buffers are used. The entire memory image is dumped starting at the specified location.

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 needs to contain 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 a LOAD 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.

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. Again, only a small part of the PAB is used. Aside from the usual information (I/O opcode and file descriptor), the PAB contains:

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

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

Execution of the SAVE command involves building the correct PAB and appending the starting and ending line number table pointers to the program image in the VDP RAM so that they can be saved also. When 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.

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 RUN command is being processed, scans the entire program segment searching for an occurrence of a DATA statement. 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 -1 (all one).

Each READ statement reads one or more items from a DATA statement. Items are separated by commas. If, during a READ, an end-of-line 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 -1, 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.

6.4 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.

6.4.1 GOTO Statement.

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.

6.4.2 GOSUB and RETURN Statements.

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, PGMPTR is restored from the entry and execution resumes there by branching to NUDEND to get the next statement.

6.4.3 ON GOSUB and ON GOTO Statements.

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 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.

6.4.4 FOR and NEXT Statements.

The FOR-TO-STEP statement and the NEXT statement are defined in conjunction with each other. The physical sequence of statement 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.

6.4.4.1 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:

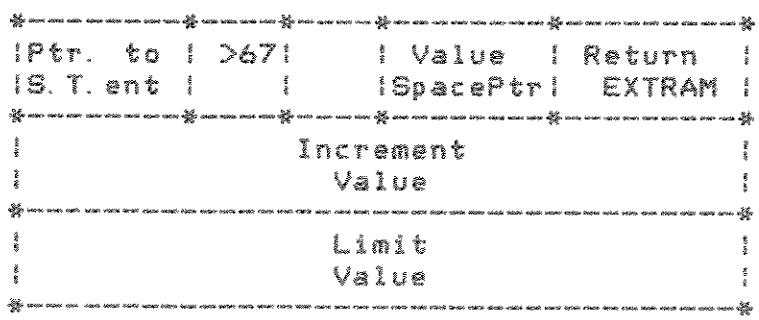


Figure 6-6 Stack-Block

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 GPL Assembler listing when following the development below.

The FOR code is entered with CHAT 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

Figure 6-7

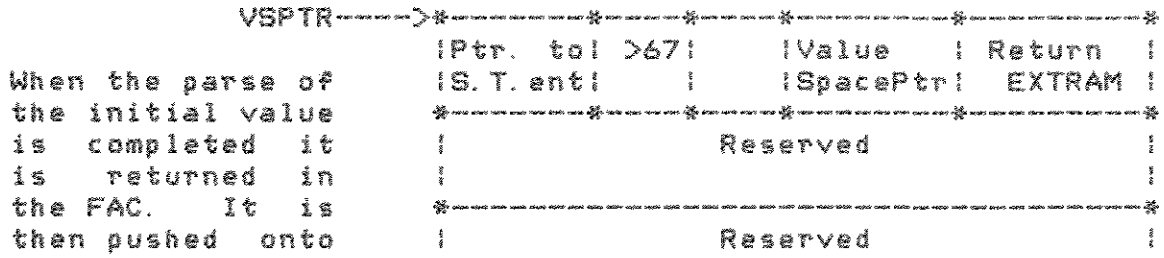
Next, SMB is called to get the pointer to the symbol's value space. The FAC entry now has the form of

Figure 6-8

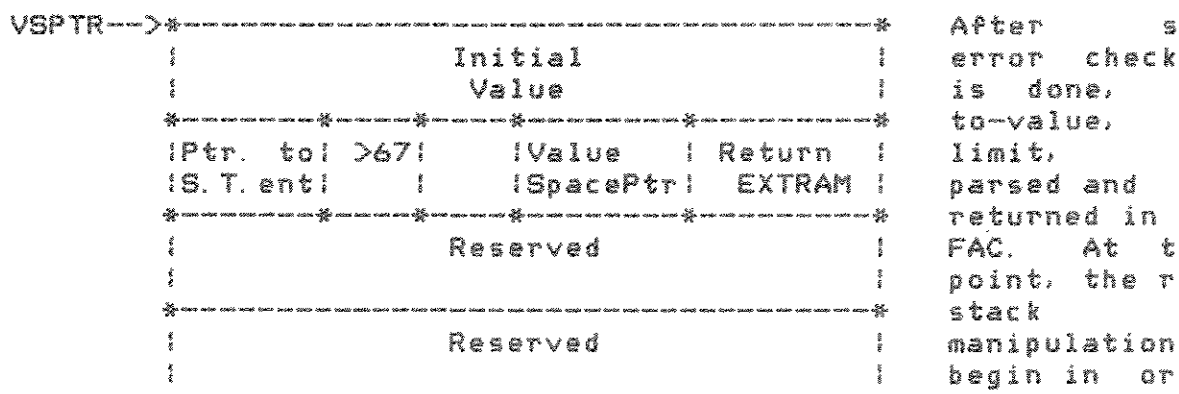
The header entry for the block is then completed with the stack FOR-block identification, >67, and EXTRAM is added to indicate to the NEXT statement where the beginning of the FOR-block is located.

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. Sixteen (16) bytes are reserved for the limit and increment and the identification entry is pushed via a XML VPUSH. A PARSE instruction is now executed to parse the next-value in the FOR statement. The stack, at the point of the parse, looks like:

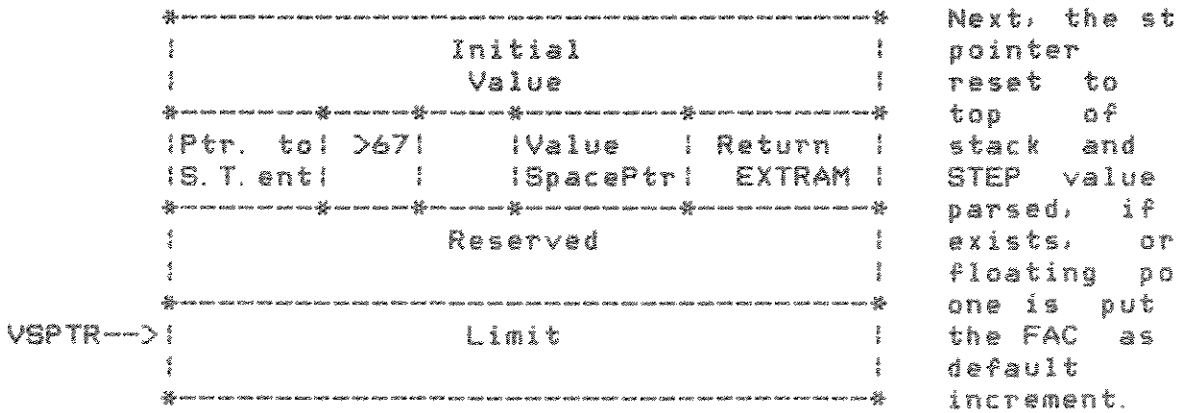


When the parse of the initial value is completed it is returned in the FAC. It is then pushed onto the stack to yield a stack that looks like:



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.

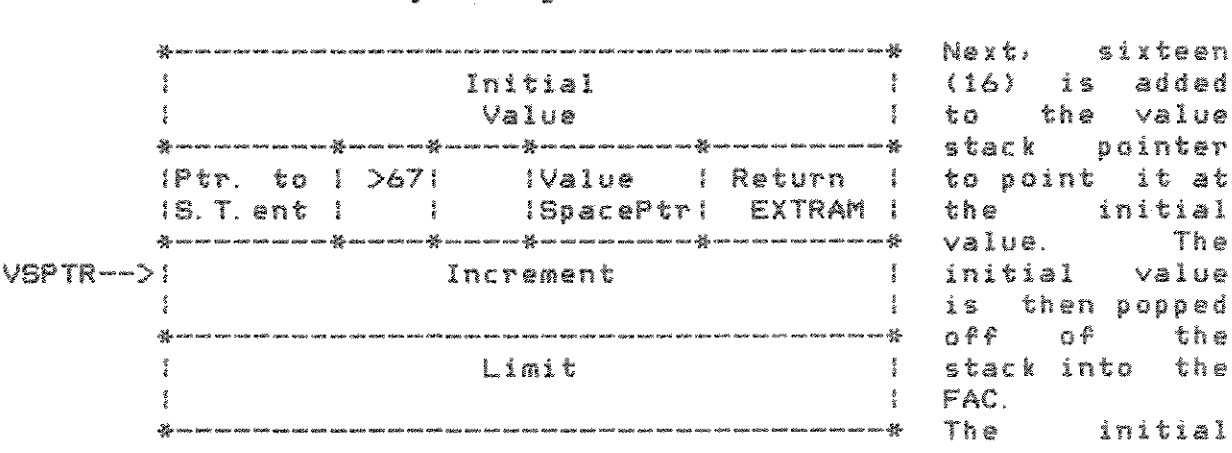
Thirty-two (32) is subtracted from VSPTR to get it pointing at the value below the FOR-block. The limit is now pushed on the stack and the stack now looks like:



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.

Twenty-four (24) 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:



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 increment entry. Eight (8) is now added to the stack pointer to get it pointing at the top of the for-block entry on the stack.

Now a check must be made to determine if the FOR-loop should be executed at all. The initial value is copied into ARG and the final value is copied into FAC and a comparison is made to see if the for-loop should be executed at all by using the floating point routine FCOMP. FCOMP 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 less than or equal to the limit if the loop is to be executed. If the increment is negative, 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 the stack entry is complete and a CONT command is executed to continue execution in the body of the FOR-loop. If the loop is not to be executed, the remaining program text is searched for the matching NEXT statement. A count 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. The FOR-block is flushed off the stack, CHAT is cleared to indicate the end of the statement has been reached and a CONT

command is executed.

6.4.4.2 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 serves as the end of a skipped block in the event that a FOR-NEXT loop is not executed. This second case has been described above.

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 sixteen (16) is subtracted (i.e., this FOR-block is discarded) from 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 index value. MOVFAC is now 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. Sixteen (16) 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 increment entry of the FOR-block on the stack.

Eight (8) is now subtracted from the value stack pointer to point at the limit and a stack comparison is made by using SCOMPB, comparing the limit on the stack with the new 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 twenty-four (24) to the value stack pointer to retain the FOR-block on the stack. We then pick up the old line number table pointer (EXTRAM) from the first entry of the FOR-block and execute a CONT command to resume execution at the top of the FOR-loop.

6.4.5 IF-THEN-ELSE Statement.

Execution of the IF-THEN-ELSE statement consists of two parts. First, the expression of the IF-clause is evaluated and second, the correct line number is selected and control flows to it. 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 If it is present, the value of the expression is checked to see if it is true and if it 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 the expression's result is false an ELSE-clause is looked for and if it is present control flows into the GOTO code otherwise it flows to the statement following the IF-THEN statement.

6.4.6 CALL Statement.

The CALL statement in the TI-99/8 BASIC is used to access subprograms written in GPL or Assembly Language. BASIC subprograms are not supported.

Execution of the CALL statement is very simple. It checks for the unquoted-string token which signifies a name and if it is present then a pointer to the name is placed in FAC+12 if the program resides in VDP RAM or the name is copied into the crunch buffer and a pointer to the crunch buffer is placed into FAC+12 if the program being executed resides in GROM. From there, the linking routine of the MONITOR is called which sends control to the GPL or ALC subprogram. When the subprogram has completed execution it returns control to BASIC by using the return routine of the monitor. When control returns to BASIC a check is made to see that the end of line has been reached. If not the subprogram did not exist or there are extraneous characters remaining on the end of the line. If the end of line has been reached a GPL CONT instruction is executed to continue executing the BASIC program or to return to top-level.

6.5 BASIC-Provided GPL Subprograms

This section describes the GPL subprograms contained in the TI-99/8 BASIC interpreter. The subprograms supported include: CLEAR, HCHAR, VCHAR, GCHAR, CHAR, KEY, SOUND, COLOR, SCREEN, GRAPHICS, MARGINS, DCOLOR, DRAW, DRAWTO, and FILL. These subprograms provide access to some of the unique features of the hardware and also make programming in BASIC easier.

6.5.1 CLEAR Subprogram.

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.

6.5.2 SOUND Subprogram.

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 begins 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 sixtieths 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.

6.5.3 COLOR Subprogram.

The COLOR subprogram is used to specify the colors of characters, blocks, or sprites, depending on the graphics mode. In Pattern mode, the COLOR subprogram is used to specify the foreground and background colors of particular character sets

(portions of the entire character set). In the Split-Screen modes, the COLOR subprogram is used to specify the foreground and background colors of individual characters. In Multicolor mode, the COLOR subprogram is used to specify the color of of a specified block. The COLOR subprogram can also be used to specify the foreground color of sprites. Execution of the COLOR subprogram 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.

6.5.4 SCREEN Subprogram.

The SCREEN subprogram is used to change the background color of the screen. In Text mode, the SCREEN subprogram is used to specify the foreground and background colors of all 256 characters. 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.

6.5.5 CHAR Subprogram.

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. Execution of the subprogram involves first parsing the character number, converting it into an integer, verifying it to be in the legal range, and saving it in a temporary location. Next, the pattern identifier is parsed. It is then converted, character by character, into hexadecimal digits. If the length of the string parsed is less than 16, the string is padded with enough zeros to make the length be 16. If the length of the string is greater than 16 it is truncated to 16. The string, once it has been converted to hexadecimal, is then moved into the appropriate place in the character table by using the character number, previously saved, as an index into the character table.

6.5.6 KEY Subprogram.

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.

6.5.7 VCHAR Subprogram.

The VCHAR subprogram is used to vertically repeat a character on the screen. In the Split-Screen and High-Resolution modes, the VCHAR subprogram cannot access the text portion of 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.

6.5.8 HCHAR Subprogram.

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.

6.5.9 GCHAR Subprogram.

The GCHAR subprogram is used to determine the character code of a character, the status of a screen pixel, or the color of a block, at a specified location on the screen. 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.

6.5.10 GRAPHICS Subprogram.

The GRAPHICS subprogram is used to specify the desired graphics mode.

6.5.11 MARGINS Subprogram.

The MARGINS subprogram is used to specify the screen margins which define a screen window.

6.5.12 DCOLOR Subprogram.

The DCOLOR subprogram is used to specify the foreground and background colors that are used in the DRAW, DRAWTO, FILL, HCHAR, and VCHAR subprograms. The DCOLOR subprogram can only be used in the graphics portion of the Split-Screen modes and in the High-Resolution mode.

6.5.13 DRAW Subprogram.

The DRAW subprogram is used to draw or erase lines between specified pixels. The DRAW subprogram can only be used in the graphics portion of the Split-Screen modes and in the High-Resolution mode.

6.5.14 DRAWTO Subprogram.

The DRAWTO subprogram is used to draw or erase lines between the current position and the specified pixels. The DRAWTO subprogram can only be used in the graphics portion of the Split-Screen modes and in the High-Resolution mode.

6.5.15 FILL Subprogram.

The FILL subprogram is used to color the area surrounding a specified pixel. The FILL subprogram can only be used in the graphics portion of the Split-Screen modes and in the High-Resolution mode.

6.6 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.

6.6.1 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).

6.6.2 STOP and END Statements.

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 EXEC 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.

6.6.3 Occurance of Breakpoints.

Breakpoints are provided in this BASIC to allow the user a strong debugging facility. This paragraph describes how a breakpoint is taken. Breakpoints can occur in three different ways.

1. A breakpoint can be set by using the BREAK command of BASIC.

2. A breakpoint can be taken by the execution of a BREAK command without a line number.

3. A breakpoint can be taken by the user depressing the FUNC-4 key on the keyboard.

Section 6.7.1 describes how a breakpoint is set.

6.7 Debugging Aids

The TI-99/8 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.

6.7.1 Breakpoints.

Breakpoints in the TI-99/8 BASIC may be caused in one of three ways. First, the FUNC-4 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 FUNC-4 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 checks to see if a GROM BASIC program is being executed and if it is the break-key is ignored and execution continues. If the break is a legal one, the screen is scrolled and the breakpoint message is displayed on the screen, as well as the current line number. The current line number table pointer (EXTRAM) is saved in the VDP RAM in a special location to indicate that a continue statement could be executed if it is entered. 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.

6.7.1.1 BREAK Statement.

The BREAK statement serves two functions in the TI-99/8 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 pointer 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 pointer is set to set the breakpoint. This method is used because the line table pointer in a VDP RAM BASIC program can never normally have an address with the most-significant bit set and no breakpoints are allowed in GROM BASIC programs. Thus, when the parser encounters a line pointer with the most-significant bit set to a one, it determines that a breakpoint has been set there and control flows to the breakpoint handler in exactly the same manner as when the FUNC-4 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 is saved and four is subtracted from it so that if a CONTINUE command is entered, the statement following the BREAK statement is where execution will begin. If this were not done the BREAK statement could never be passed, as a CONTINUE would cause the BREAK statement to be executed, putting the interpreter into an infinite loop, as long as the user continues to enter a CONTINUE command.

6.7.1.2 UNBREAK Statement.

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 pointers 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 pointer will make no changes to the pointer 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 pointer is reset for each line listed. If a line is listed which does not exist, a warning message is issued, but, the remainder of the line list is searched.

6.7.2 CONTINUE Command.

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 was saved in the VDP by the breakpoint handler) been taken, the program flag is switched back to program mode, CONTINUE is disabled (the special VDP RAM continue word is cleared), and control flows into the code which executes the GPL EXEC instruction to start execution again. 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.

6.7.3 Tracing.

The trace feature of the TI-99/8 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 a RTNB instruction is executed to return to the parser to execute the statement, for which the line number has just been displayed.

6.7.3.1 TRACE Statement.

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

6.7.3.2 UNTRACE Statement.

Execution of the UNTRACE statement involves simply clearing the trace flag bit (bit 4) of the interpreter variable, FLAG, and executing a GPL CONT instruction to return control to the parser.

SECTION 7

GROM BASIC PROGRAM SUPPORT

BASIC programs placed in GROM have only limited access to the BASIC interpreter. The BASIC editing facilities cannot be utilized as it does not make sense to try to edit a ROM any way. BASIC programs contained in GROM cannot be directly accessed from BASIC. They must be invoked either by selecting them from the main menu or by some routine selected from the main menu invoking the. There are also certain restrictions placed upon BASIC programs which reside in GROM due to some bugs contained in the interpreter that have not been fixed due to lack of memory space to fix them. A GROM BASIC program cannot use READ and DATA statements as the RESTORE command does not work for GROM BASIC programs. Also, the CALL CHAR subprogram cannot be used to specify the character fonts of any special characters. The CALL CHAR subprogram can be used with the null string specified as the second argument to allocate the space and then a custom GPL subprogram can be written to place all of the character fonts in the reserved area.

7.1 VDP Use

The VDP RAM is utilized in exactly the same way with GROM BASIC programs as with VDP BASIC programs except that two things differ. First, the program obviously does not reside in the VDP, thus leaving more space for the symbol table and string space. Second, the symbol names must be stored in the symbol table along with the symbol table entry so that the symbol table searching routine will work properly. This condition is taken care of automatically by the prescan portion of the interpreter.

7.2 Program Representation

GROM BASIC programs are represented within a GROM exactly as they would appear in the VDP as described in section 3.2. The line number table, instead of appearing at a lower address than the program text appears at a higher address. This due to the manner in which the program image is built by the BASGROM conversion program. Also, the program is set up with a link in the GROM header and a startup procedure that is invoked by

selecting the program from the main menu. This is how access is gained to a GROM BASIC program.

7.3 Execution Sequence

A GROM BASIC program is treated exactly the same as a VDP BASIC program except that the entry point into BASIC for the GROM program sets a flag in the interpreter indicating that the program resides in GROM and the text is to be read from there. This flag is checked in such places as PGMCHR (gets next program token), the string routines (where to get string constants from), the prescan routine (to put names in the VDP), the error handler, the breakpoint handler and the normal end of execution handler. The flag either indicates where to read text from or where to return to in the case of an error or the normal end of execution.

SECTION 8

WRITING GPL SUBPROGRAMS

The TI-99/8 BASIC supports subprograms written in GPL. GPL subprograms may be accessed by execution of the BASIC CALL statement with the subprogram taking care of picking up any arguments and assignment of values returned to variables in the BASIC program. GPL subprograms may be used to achieve a speed increase over the same code written in BASIC or to utilize some of the system resources that are difficult to access from BASIC. Examples of these are menu processing and random screen access, respectively.

8.1 Execution Sequence

The execution of a GPL subprogram may be divided into five major sections. First, the GPL subprogram must take care of setting up the linkage from BASIC so that control may return to the interpreter. This involves a call to PGMCTR to set up the text pointer (see section 4.2.1). Second, the subprogram must take care of parsing any arguments that it may need by using the PARSE statement and doing some syntax checking (see section 7.2.2). Third, the subprogram must execute whatever function it is designed to do. Fourth, the subprogram must take care of assigning any values to be returned to BASIC variables. And, fifth, the subprogram must return control to the interpreter by calling RPL to take care of the linkage (see section 7.2.1).

8.2 Useful Subroutines

Routines are provided to make programming GPL subprograms easier and to achieve portability between future models of personal computers. These routines provide functions in the areas of:

* Linkage to BASIC
* Parameter acquisition and returning values to the BASIC program
* Handling string space

* Manipulating the floating point stack

8.2.1 Linkage to BASIC.

* PGMCTR
A GPL subprogram must call PGMCTR to set up for the processing of parameters.

```
PGMCTR EQU  >42          ADDRESS IN MONITOR
START  CALL PGMCTR       SET UP BASIC POINTERS
```

This subroutine sets up memory location PGMPTR(>2C-2D) and places the first byte after the subprogram name into CHAT(>42). This will be the "(" token (>B7) if a valid parameter list is included or >00 if no parameter list is included. The subprogram should validate the "(" token or >00 as part of standard syntax checking.

* RPL
A GPL subprogram returns to execute the next BASIC statement with a CALL to RPL.

```
RPL  EQU  >12            EQUATE IN MONITOR
END  CALL RPL            RETURN TO BASIC PROGRAM
*    DOES NOT RETURN
```

Although this routine does not return, a CALL must be used rather than a Branch instruction. If the subprogram processed the parameter list properly and there was no syntax error, CHAT will contain a >00. This will be checked by BASIC after the CALL to RPL.

* ERR
Errors can be reported to the user with the ERR subroutine.

```
ERR  EQU  >1C            EQUATE IN MONITOR
ERR1 CALL ERR            CALL ERROR PROCESSOR
     DATA #MSG1          MESSAGE NUMBER 1
*    DOES NOT RETURN

     BASE 0,0,>300,>300,0,0,>60
MSG1 DATA 19,:INCORRECT STATEMENT:
     BASE 0,0,>300,>300,0,0,0
```

The ERR routine does not return. The message pointed to by the data word is displayed and the BASIC program execution is terminated. The first byte of the message is the length of the message string which follows. The

message text must be biased by >60. This is accomplished with the BASE statement shown.

* WARN
Warning message can be reported to the user with the WARN routine.

```
WARN EQU  >1A          EQUATE IN MONITOR
W1   CALL WARN         ISSUE WARNING MESSAGE
     DATA #MSG1        MESSAGE 2

     BASE 0,0,>300,>300,0,0,>60
MSG1 DATA 13,:UNIT NOT OPEN:
     BASE 0,0,>300,>300,0,0,0
```

The text "* WARNING-" followed by the specified message is displayed. The WARN routine then returns to the GPL subprogram. The message is specified as in the ERR routine(see above).

8.2.2 Parameter Acquisition.

Individual characters and tokens can be tested by comparing CHAT with the expected value. The next character in the parameter expression can be placed into CHAT with the XML PGMCHR. The value of a parameter expression can be computed with the PARSE statement. The address of a variable name parameter can be determined with the XML SYM and XML SMB commands. A value can be assigned to a variable with the XML ASSGNV. The routines XML VPUSH and XML VPOP are used to manipulate the floating point stack.

* PGMCHR
The next character of the parameter list can be obtained with the XML PGMCHR. The next character is placed in CHAT(>42). The beginning of a GPL subprogram could be:

```
PGMCHR EQU  27          XML EQUATE
LPAR$  EQU  >B7         LEFT PAREN TOKEN VALUE
START  CALL PGMCTR      SET TO SCAN PARM LIST
       $IF  @CHAT .NE. LPAR$ GOTO ERR1  SYNTAX
       XML  PGMCHR      GET TOKEN AFTER LPAR
```

* PARSE
An expression (string or numeric) in the parameter list can be evaluated with the PARSE statement. The operand of the PARSE statement is the precedence at which to stop parsing. If the parser encounters an unmatched token of

the specified precedence or higher, the evaluation will stop and control will return to the subprogram. When the PARSE statement is executed the first character of the expression must be in CHAT. To parse and stop at the end of a parameter list or a comma the following is used.

```
COMMA$ EQU >B3          COMMA TOKEN
RPAR$  EQU >B6          RIGHT PAREN TOKEN
       PARSE RPAR$      PARSE THE EXPRESSION
       $IF @CHAT .NE. COMMA$ GOTO ERR1
```

The PARSE statement will evaluate an expression until a ")" or higher precedence token is encountered at the same level. Note that other operators in an expression can cause this PARSE to be temporarily stacked(e.g. a "(" will do a PARSE to a ")" and then return to the original level). All PARSE statements must appear in GROMs 0, 1, 2 or 3 as the BASIC interpreter cannot execute a PARSE statement from GPL with the top bit of the address set to a one (GROM 4 begins at >8000).

8.3 Restrictions

GPL subprograms have a great number of restrictions placed upon them by the way in which the BASIC interpreter utilizes memory. GPL subprograms may not use any of the CPU RAM from >18 to >49 nor can they change CPU >6E-6F without using the XML VPUSH and XML VPOP commands. VDP RAM cannot be used in any manner without using one of the three methods described in the following three sections. The GPL PARSE statement may not be used in a GPL subprogram except within GROM 3. Sprites may not be used from a GPL subprogram called from BASIC.

8.4 Stealing VDP RAM

VDP RAM may be "stolen" or used by an applications program in several ways, depending upon the amount of memory needed and and the length of time it is needed.

8.4.1 Start Up.

VDP memory may be permanently allocated to an applications program at start up time by taking the top (highest adressable area) of VDP RAM and "faking out" BASIC so it doesn't know that

memory exists. This is done with the CALL PREP GPL subprogram which can be called from the GPL start-up portion of a BASIC applications program. A listing of PREP is given in Appendix B. After the applications program has been selected on the main menu of the computer, a call to PREP will allocate the number of bytes specified by the first two bytes of the FAC to the applications program by changing the value of MAXMEM (>70). Thus, BASIC will never know the bytes exist and therefore not try to use them.

Once the BASIC program has started execution the reserved area can be accessed by other GPL subprograms.

8.4.2 MEMCHK.

Memory may be allocated in a less permanent manner than using CALL PREP by using a CALL MEMCHK. MEMCHK may be called only after the BASIC program has begun execution. MEMCHK allocates a block of memory specified by the two-byte count provided in FAC (CPU >4A-4B). The block of memory is allocated between the BASIC symbol table and the string space (see figure 3.4.1). This memory will remain allocated while the BASIC program is executing but is destroyed whenever the BASIC program completes execution, an error occurs, any BASIC program residing in VDP RAM is edited and when BASIC is exited via the BYE command.

8.4.3 String Space.

VDP RAM may be temporarily "borrowed" by allocating a temporary string in the string space. The string is specified by the byte count provided in BYTE (CPU >0C,0D) and may be anywhere between 0 and 255 bytes, inclusive. To allocate the bytes a call to GETSTR is made to get the string. GETSTR will allocate a string of the specified number of bytes if memory is not full otherwise an error will occur after a garbage collection is performed and another attempt is made to allocate the bytes. A pointer to the allocated bytes is returned by GETSTR in SREF (CPU >1C,1D). The "string" allocated is of the form described in section 3.7.

The temporary string may be used until a garbage collection is performed as it is or may be made more permanent by building a temporary string FAC entry and pushing it onto the value stack using VPUSH. VPUSH will set the backpointer pointing into the stack, making the string semi-permanent. A string made semi-permanent in this manner can be freed by popping the entry off of the value stack by using VPOP.

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	undefined	AC
undefined	AD	INTEGER	AE
REAL	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

Keyword	Internal Value	Keyword	Internal Value
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
MAX	DF	MIN	E0
RPT$	E1	VALHEX	E2
FREESPACE	E3	TERMCHAR	E4
undefined	E5	ALPHA	E6
LALPHA	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 parentheses 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

System Flags

This appendix describes all the system flags. The meaning of each flag, its location, the programs in which it is used, and a description of each used bit will be included.

B.1 WARN$$

WARN$$ checks the special warning handling conditions which can be set by an ON WARNING statement and does the following based upon those conditions:

ON WARNING PRINT - Prints and continues execution.
ON WARNING STOP - Prints and stops.
ON WARNING NEXT - Ignores the warning and continues.

The flag byte for warnings is WRNFLG. It is located at >8387 and is used in WARNINGS and by "ON WARNING". The meaning of bits in WRNFLG is as follows:

Bit	Reset	Set
0	WARNING PRINT	PRINT OFF
1	WARNING NEXT	STOP
2	BREAK ALLOWED	BREAK NOT ALLOWED

APPENDIX C

GPL10 As A Debugging Aid

Experience with the TI-99/8 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 TI-99/8. It has a customized GPL interpreter as well as several support modules which provide the interface between the emulator and the 990/10 operating system.

A slightly modified assembly language (TI-99/8 ROM) code is link-editted in with the customized GPL interpreter and support routines to use the 990/10 debugging facilities. The modifications needed in the assembly language code are described in a later section.

When the GPL10 emulator is invoked (with the GPL10 SCI command on the 990/10) 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 TI-99/8.

The CPU RAM is allocated first in the 990's memory, then the VDP RAM and finally the GROM. The SCI GPL10 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 whatever GPL code is to be debugged. This file should, although it is not required, include BASIC.

C.1 Assembly Language

In order to use GPL10, a special link file must be created for the link-editor. This file should look something like:

```
TASK GPLINT
LIBRARY     .SCI990.S$OBJECT
INCLUDE VOL.GPL10.OBJ.MAIN
INCLUDE VOL.GPL10.OBJ.CRT
INCLUDE VOL.GPL10.OBJ.DEBUG
INCLUDE VOL.GPL10.OBJ.FLTPT990
INCLUDE VOL.GPL10.OBJ.CSN990
INCLUDE VOL.GPL10.OBJ.TRINSIC
INCLUDE VOL.GPL10.OBJ.BASSUP
INCLUDE VOL.GPL10.OBJ.PARSE
INCLUDE VOL.GPL10.OBJ.SUBPROG
INCLUDE VOL.GPL10.OBJ.DATA
SEARCH
INCLUDE VOL.GPL10.OBJ.EMULATOR
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 or VDP RAM to run properly, conditional assemblies and/or MACROs are needed. This is due to the differences in accessing the 990's continuous memory and GROM or VDP chips.

C.2 Accessing GROM

The following is a typical example of how the GROM chips are 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 GPL10 by:

```
MOV  @ADDR,R13             Use R13 for common purpose
AI   R13,GROM              GROM offset in memory
MOVB *R13+,@BYTE           READ a byte
```

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

C.3 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 chip:

```
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 GPL10's "VDP" area is written as:

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

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

In order to write to the VDP chip 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 GPL10:

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

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

C.4 GPL Code

The only thing that must be done to correctly execute GPL code on GPL10 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 GPL10 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 TI-99/8 are available on the 990/10, but, any characters put on the screen of the TI-99/8 will appear on the 990/10's CRT in some form or another. A TI-99/8 console or simulator is necessary to display the correct characters on the screen.