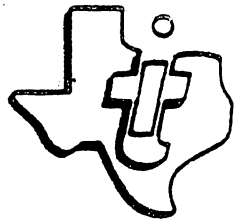
The Engineering Staff of
TEXAS INSTRUMENTS INCORPORATED
Semiconductor Group



CONFIGURABLE POWER BASIC REFERENCE MANUAL

FEBRUARY 1979
MP 318

TEXAS INSTRUMENTS
INCORPORATED

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time to improve design and to supply the best possible product for the spectrum of users.

Configurable POWER BASIC (Part Number TMSW510F) is copyrighted by Texas Instruments Incorporated, and is sole property thereof. The software may not be reproduced in any form without written permission of Texas Instruments Incorporated. However, output from the Configurator may be reproduced for resale exclusively by the customer purchasing the Configurable POWER BASIC Package.

All manuals associated with Configurable POWER BASIC (MP 318) are printed in the United States of America and are copyrighted by Texas Instruments Incorporated. All rights reserved. No part of these publications may be reproduced in any manner including storage in a retrieval system or transmittal via electronic means, or other reproduction in any form or by any method (electronic, mechanical, photocopying, recording, or otherwise) without prior written permission of Texas Instruments Incorporated.

Information contained in these publications is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor for any infringement of patents or rights of others that may result from their use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

TABLE OF CONTENTS

SECTION I. INTRODUCTION

SECTION II. POWER BASIC PRIMER

SECTION III. GENERAL PROGRAMMING INFORMATION

SECTION IV. BASIC COMMANDS

SECTION V. BASIC STATEMENTS

SECTION VI. CHARACTER STRINGS

SECTION VII. POWER BASIC FUNCTIONS

SECTION VIII. POWER BASIC CONFIGURATION PROCESS

SECTION IX. INSTALLATION OF TARGET POWER BASIC APPLICATION

APPENDICES

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF TABLES (continued)

SECTION I

INTRODUCTION

1.1 GENERAL

POWER BASIC* is a family of software products offering a wide range of features and capabilities. These products are available in a variety of forms including ROM's, TM990 boards, and floppy diskettes, in addition to the programming features normally found in BASIC**. POWER BASIC offers the user features specifically designed to support real-time industrial control applications. Two family members, Evaluation Basic (TM990/450) and Development Basic (TM990/451 and TM990/452) are designed to execute in a TM990/100M or TM990/101M microcomputer board environment. The third member of the family is Configurable POWER BASIC (TMSW510F), utilizing the higher performance FS990 floppy disk-based system with either the 911 or 913 Video Display Terminal for program development. Programs developed on either Evaluation or Development BASIC are completely upward compatable to Configurable POWER BASIC. The FS990 system provides a convenient, efficient environment for the preparation of algorithms, intended for and configured into, customized POWER BASIC ROMs to be executed in the TM990 board based application. Figure 1-1 presents the FS990 minicomputer environment in which Configurable POWER BASIC executes.

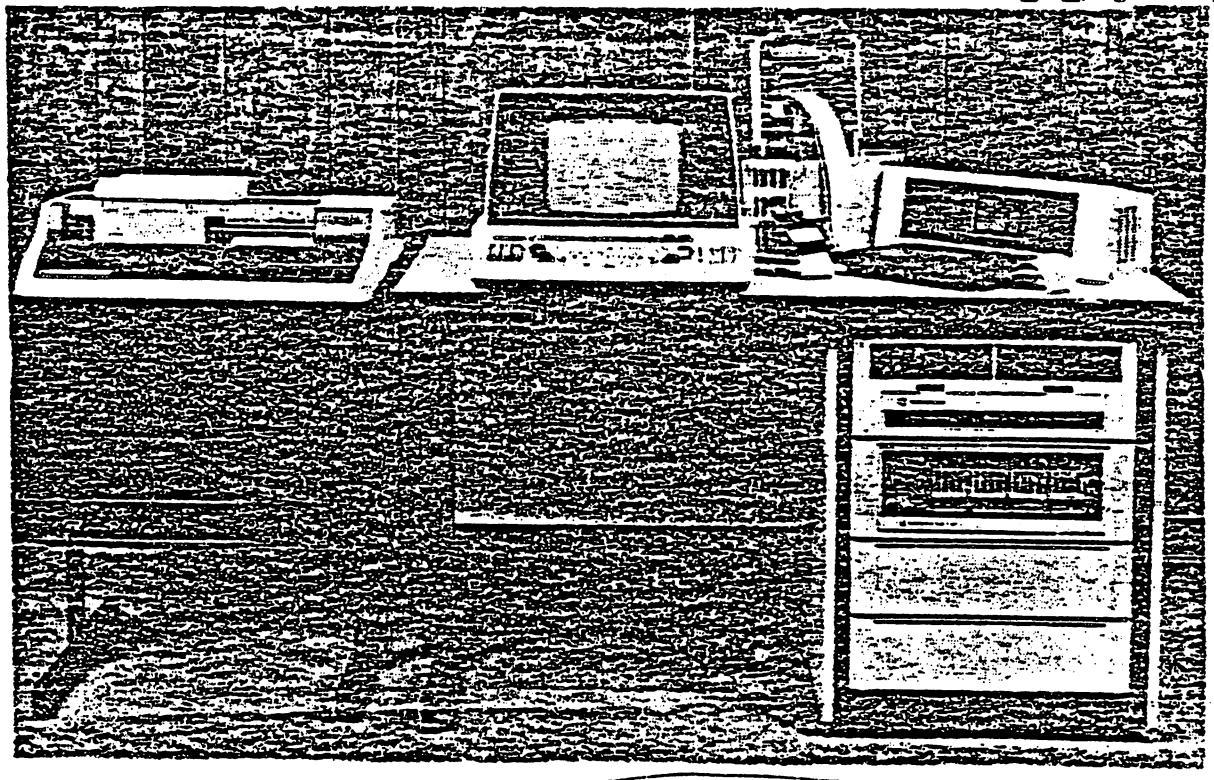


FIGURE 1-1. DEVELOPMENT AND CONFIGURATION CYCLE

* Trademark of Texas Instruments
  See POWER BASIC REFERENCE MANUAL #MP308 for more information
** Trademark of Dartmouth University

1.2 POWER BASIC OVERVIEW

The POWER BASIC language is easy to use and understand, and eliminates the need to use assembly language in writing an application program. The user may enter program statements or commands required to examine, debug, or run a program. Each statement or command is completed with a carriage return which terminates and enters the line. POWER BASIC automatically advances one line position and waits for additional keyboard input. Once the program is finished to the user's satisfaction, the POWER BASIC Configurator may be executed to produce a smaller and faster ROM/RAM partitioned POWER BASIC interpreter which may be stored in ROM for later insertion in the TM990 board system. The POWER BASIC Configurator scans the user's BASIC application program, "remembering" via a control file, each of the POWER BASIC Statements and Functions used. The user then executes the TX990 LINK Editor, using the Configurator-generated Link control file to complete the configuration process.

1.3 POWER BASIC FEATURES

POWER BASIC has many features that make it ideal for use in the industrial control environment:

- 48 bit floating point arithmetic
- 24 hour time of day clock
- elapsed time for 1/25th of a second or greater
- asynchronous transfer to user-specified statement number on user selected interrupts
- string manipulation capabilities
- 3 character variables
- sophisticated editor
- calls to assembly language routines directly from POWER BASIC
- complete FS990 file management from POWER BASIC
- Configurator minimizes memory requirements of any POWER BASIC program

The benefits the user will derive from Configurable POWER BASIC can be seen when the configuration process is complete. The resulting run-time module is stand-alone. It is customized at the BASIC statement and function level with the specified POWER BASIC application program, resulting in a faster and much smaller user application than with conventional BASIC interpreters. Further, this run-time module is appropriately ROM/RAM partitioned for exectution in a TM990/100M or TM990/101M microcomputer board system, and can be tested thoroughly with the FS990 system through in-circuit emulation using the AMPL* Microprocessor Prototyping Lab.

1.4 DEVELOPMENT AND CONFIGURATION CYCLE

Figure 1-2 depicts the steps involved in POWER BASIC program development during the Configuration process. Most programs can be completely developed and debugged on the Host POWER BASIC Interpreter which executes on the FS990 system. Additional debugging may be performed using Development BASIC in a TM990 board system when interface to the specific application is required. The Configurator scans the application program and "remembers" each of the statements and functions used in it. It then 1) produces a corresponding Link Control file to selectively include the referenced POWER BASIC statement and function routines, 2) produces the "ROOT module" containing the POWER BASIC application program, and 3) produces a listing for a hard copy of the Configuration process. The execution of the TX990 Link Editor using the Link file generated by the Configurator produces the final "customized" Target POWER BASIC Interpreter. The resultant Linked Object module may then be programmed into EPROM's using the PROM programmer available on the development system.

Note that throughout this manual the term "HOST POWER BASIC INTERPRETER" refers to the BASIC Interpreter which executes on the FS990 system, while "Target" POWER BASIC Interpreter refers to the BASIC Interpreter generated by the configuration process and Linkage Editor. Also note that some environment dependent POWER BASIC statements and functions operate differently on the Host POWER BASIC interpreter than on the Target POWER BASIC Interpreter. These environmental differences are documented in the appropriate paragraphs of Section III through VII of this manual.

A full discussion of installation procedures for the Host POWER BASIC Interpreter can be found in Section II. Section VIII contains a detailed description of the Configurator and the configuration process.

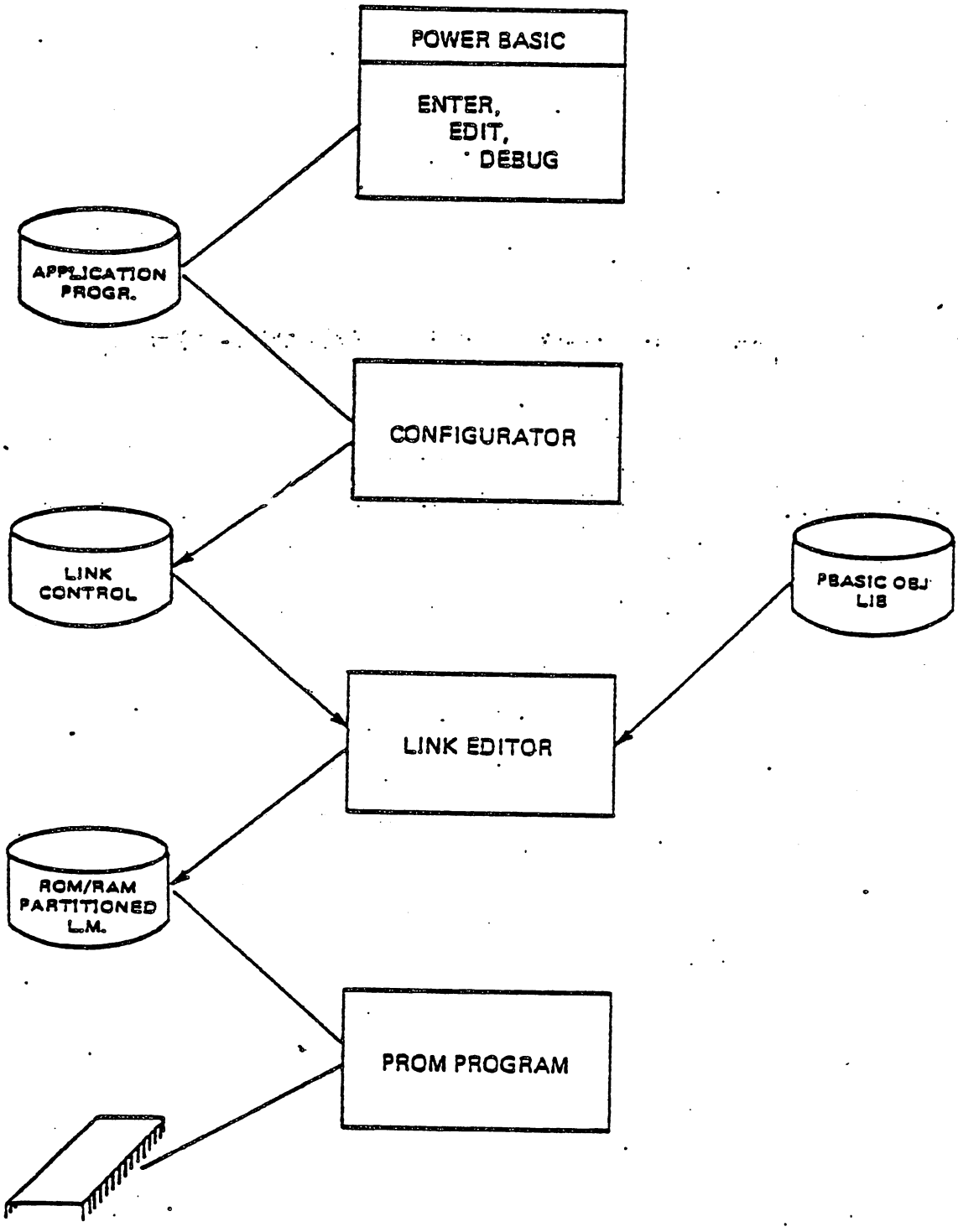


FIGURE 1-2. CONFIGURABLE POWER BASIC ON FS990/4

1.5 CONVENTIONS USED IN THIS MANUAL

The following conventions are used to describe the statements, commands, and examples in this manual:

Numeric values for command parameters are decimal unless otherwise specified.

Angle brackets (<>) indicate essential elements of user-supported data in statements, commands, and examples.

```
10 LET <variable> = <expression> to <expression>
```

Braces ({}) indicate a choice between two or more possibilities (alternative items), one of which must be included.

```
10 ON {<variable>   } THEN GOSUB <statement number list>
      {<expression> }
```

Brackets ([]) enclose optional items.

```
10 [LET] A=4*ATN(1)
```

Items in capital letters must be entered exactly as shown.

Items in lower case letters are user-supplied characters.

SECTION II

LOADING AND EXECUTION OF HOST POWER BASIC INTERPRETER AND CONFIGURATOR

2.1 INTRODUCTION

The Configurable POWER BASIC package consists of two modular software packages shipped on two diskettes. These diskettes contain the HOST POWER BASIC INTERPRETER and the POWER BASIC CONFIGURATOR. The modules contained on these diskettes differ only in the terminal used for the system console.

The HOST POWER BASIC INTERPRETER provides the capability for extended POWER BASIC program development and debug, with full floppy disk and file support.

The POWER BASIC CONFIGURATOR produces the link control file and root module that are required to generate the customized POWER BASIC interpreter. In addition, the startup module and a library of POWER BASIC object modules to be used by the LINK EDITOR are included on these diskettes.

The two diskette disk names and the pathnames of their contents are listed as follows:

DISKETTE NAME	CONTENTS
CONFIGURABLE POWER BASIC VDT911	:CBASIC/V11 :CONFIG/V11 :CBASIC/LIB :STARTC/OBJ
CONFIGURABLE POWER BASIC VDT913	:CBASIC/V13 :CONFIG/V13 :CBASIC/LIB :STARTC/OBJ

To load and execute either the POWER BASIC or Configurator software, the user must provide a version of the TX990 Operating System (release 2.3 or later). The user then selects the package(s) which match the device type of the system console supported by the operating system.

Either of the selected POWER BASIC or Configurator packages may then be loaded into the dynamic task area of the user-supplied TX990 Operating System. Note that both software packages cannot concurrently reside in the dynamic task area since both packages fully utilize the remaining task area for user program storage. These packages may be installed (loaded) into the dynamic task area and executed under control of either 1) the Operator Communications Package (OCP), or 2) the Terminal Executive Development System (TXDS). The following paragraphs describe the loading procedure for each method, the LUNO's required by each package, and the operating system support required in the user's TX990 Operation System.

2.2 LOADING OF POWER BASIC AND CONFIGURATOR PACKAGES

Both POWER BASIC and the Configurator may be loaded and executed using either the OCP module or the TXDS control program, (whichever is supported by the user's TX990 Operating System). When using the OCP MODULE, perform the procedures itemized in Paragraph 2.2.1. When using the TXDS control program, perform the procedures itemized in Paragraph 2.2.2.

Note that two POWER BASIC packages and two Configurator packages have been shipped on the two diskettes. The two software packages differ only in the device used for system console: 1) the 911 Video Display Terminal, or 2) the 913 Video Display Terminal. The user must select and load the software package whose console matches the system console in the TX990 Operating System being used. The two packages are supplied on the diskettes under the pathnames presented in Paragraph 2.1. The pathname extensions designate the system console device type for these modules (e.g.,/V13 designates the 913 Video Display Terminal as the console device of that module).

2.2.1 LOADING USING OCP

Proceed as follows:

1. Load the TX990 Operating System that meets the operating system requirements as presented in Paragraph 2.4 by performing the steps in "Loading the Operating System", (Section II of the TX990 OPERATING SYSTEM PROGRAMMER'S GUIDE).

2. Press the exclamation point (!) key on the system console keyboard.

3. If OCP is included in the system, it responds with a period (.) prompt:

```
.
```

4. Place the diskette containing the selected POWER BASIC (or Configurator) object module into the left diskette drive.

5. Using OCP's LP (Load Program) command, load the selected object module from the diskette media into memory as follows:

`.LP,DSC:CBASIC/***`	Load from floppy diskette DSC the object module file CBASIC/***, where *** is the console device (V11 or V13).

similarly

`.LP,DSC:CONFIG/***`	Load from floppy diskette DSC the object module file CONFIG/***, where *** is the console device (V11 or V13).

6. Enter OCP's EX (Execute) command to execute the POWER BASIC (or Configurator) object module and terminate OCP as follows:

`.EX;10.TE.`	Executes POWER BASIC (or Configurator) and terminates OCP.

7. Observe the following printout or display on the system console:

```
CONFIGURABLE POWER BASIC REV FS.n.m
*READY
```

or similarly,

```
POWER BASIC CONFIGURATOR REV C.n.m

APPLICATION SRCE=
```

where,

n = release number
m = revision number

8. At this point the POWER BASIC (or Configurator) package will be executing and ready for your input.

2.2.2 LOADING USING THE TXDS CONTROL PROGRAM

Proceed as follows:

1. Load the TX990 Operating System that meets the operating system requirements as presented in Paragraph 2.4 by performing the steps in "Loading the Operating System" (Section II of the TX990 OPERATING SYSTEM PROGRAMMER's GUIDE).

2. Press the exclamation point (!) key on the system console keyboard.

3. If OCP is included in the system, it responds with a period (.) prompt:

```
!
.
```

If OCP is not included, proceed to step 5.

4. Execute the TXDS Control Program by responding to the period (.) prompt as follows:

```
!
. EX,16.TE.
```

5. Observe the following printout on display presented on the system console:

```
TXDS 936215 ** 010/77 2:05

PROGRAM:
```

6. Place the diskette containing the selected POWER BASIC (or Configurator) object module into the left diskette drive and then enter the pathname of the POWER BASIC (or Configurator) object module in response to the "PROGRAM:" prompt as follows:

PROGRAM:DSC:CBASIC/***	Load and execute the object module file CBASIC/***, where *** is the console device (V11 or V13).

or similarly,

PROGRAM:DSC:CONFIG/***	Load and execute the object module file CONFIG/***, where *** is the console device (V11 or V13).

7. Depress the carriage return key and observe that the INPUT: prompt is printed out or displayed on the sytsem console.

NOTE

The asterisk (*) feature can be used in lieu of the carriage return/NEW LINE entry to by pass the remaining prompts (Input, Output, Options).

8. Make a null entry by depressing the carriage return and observe that the OUTPUT: prompt is printed out or displayed on the system console.

9. Make a null entry by depressing the carriage return and observe that the OPTIONS: prompt is printed out or displayed on the system console.

10. Make a null entry by depressing the carriage return and the POWER BASIC (or Configurator) program loads from diskette and begins execution. Observe the following printout or display on the system console:

CONFIGURABLE POWER BASIC REV FS.n.m

*READY

or similarly,

POWER BASIC CONFIGURATOR REV C.n.m

APPLICATION SRCE=

where,

n = the release number
m = the revision number

11. At this point the POWER BASIC (or Configurator) package will be executing and ready your input.

2.3 LUNO's

TX990 uses logical unit numbers (LUNOs) to represent devices ar files. POWER BASIC performs I/O to a LUNO. This is converted by th operating system to represent the physical device on file to which th LUNO is assigned.

POWER BASIC uses LUNO 0 and LUNO's 20 through 24 . Th Configurator uses LUNO 0 and LUNO's 21 and 22 . LUNO 0 is assigne to the system console by the TX990 Operating System. Al communications between the user and a task (i.e., POWER BASIC th Configurator) are performed through LUNO 0.

LUNO's 21 and 22 are assigned by the Configurator to the device or diskette files which are to receive the root module, link contro file, and listing outputs. LUNO's 21 through 24 are assigned b POWER BASIC to the devices or diskette files referenced by the user's POWER BASIC program or during user program development.

All POWER BASIC output is initially directed to the system console device, however the user has the option to redirect all output to any other supported output device or diskette file by executing the SPOOL and UNIT POWER BASIC statements.

When POWER BASIC or the Configurator are exited to return to the Operating System, all files are closed.

2.4 OPERATING SYSTEM REQUIREMENTS

The Host POWER BASIC Interpreter and Configurator software packages utilize the TX990 Operating System provided by the user to load, execute, and perform file and device I/O. Therefore certain requirements must be specified for the user's operating system. Operating Systems which meet the requirements must:

- be version 2.3 or later
- provide either OCP or TXDS for program load and execution
- provide full diskette file management
- provide full task support
- have at least 4 file LUNO blocks specified during system generation

Optionally, the user's TX990 Operating System may also support a line printer and 733 ASR terminal and cassettes. These devices are useful with both the POWER BASIC and Configurator packages.

2.5 EQUIPMENT REQUIREMENTS

The POWER BASIC and Configurator software packages require the following hardware components connected in the standard FS990 minicomputer configuration:

- FS990 with 24K words of RAM
- Model FD800 dual floppy disk drive
- Console terminal (911 VDT or 913 VDT)

The following optional equipment may also be supported:

- 733 ASR cassettes and printer
- Model 810 line printer

2.6 RECOMMENDED PROCEDURE

Before the TI-supplied diskettes are used, they should be copied onto backup diskette(s) and the masters stored in a safe place. This ensures that the master diskettes are available if the backup diskette(s) are destroyed. The diskette/disc backup and initialize program will be used to copy the diskette as outlined below. For details on BACKUP, refer to Section X of the Model 990 Computer TX990 Operating System Programmer's Guide.

The recommended procedure is:

1) Use only one backup (or production) diskette when executing either the Host POWER BASIC Interpreter or the Configurator.

2) Select the configurable POWER BASIC diskette whose console device matches the system console of the TX990 Operating System being used.

3) Copy the appropriate master diskette, "Configurable POWER BASIC VDT9XX" to the backup diskette.

4) Copy the TX LINK Editor (modules :TXSLNK/SYS, :TXLOVM/SYS, :TXLOVL/SYS) from TXDS LINK EDITOR DISKETTE to the backup diskette.

NOTE

Version 2.3.1 or later of the Link Editor is required to accept the link control file produced by the Configurator.

The backup (or production) diskette will then contain the appropriate POWER BASIC and Configurator object modules for use with the available TX990 Operating System. It will also contain all required modules and control programs to complete generation of a configured POWER BASIC application. The backup diskette in its final form will then contain the following files:

:CBASIC/***	Object modules whose console device matches Operating System being used. (*** is the file extension V11 or V13.
:CONFIG/***	
:CONFIG/LIB	Object Library and STARTC modules referenced by link control file produced by Configurator.
:STARTC/OBJ	
:TXSLNK/SYS	TX2.3.2 Link Editor and associated overlays.
:TXLOVM/SYS	
:TXLOVL/SYS	

NOTE

The Link Editor produces temporary files on the diskette from which the Link Editor was loaded during its execution. Therefore, if an exceptionally large application program is to be linked, the user may need to transfer the Configurator Object Library module (:CONFIG/LIB), start module (:STARTC/OBJ), and the TX Link Editor (with all overlays) to a separate diskette. The Link Edit may then be performed using the copied diskette space for all Link Editor temporary files.

2.7 DEBUG CHECKLIST

If the POWER BASIC or Configurator packages do not correctly respond with the banner message as presented in Paragraphs 2.2.1 and 2.2.2, the user should verify the FS990 system against the following checklist.

- Check TX990 Operating System
  - Verify that Operating System console device corresponds to console terminal being used.
  - Load and execute a sample TXDS program (e.g., TXCCAT) to verify system operation
- Check POWER BASIC and Configurator modules
  - Verify that the system console of TX990 Operating System

matches the console device of the selected POWER BASIC and Configurator software packages.
  - If the modules were copied to a backup diskette, verify the copy, and try loading and executing the master diskette object modules.

- Check parameters of TX990 Operating System
  - Verify that the TX990 Operating System meets all specifications as required by the POWER BASIC or Configurator packages as presented in Paragraph 2.4.

If the cause of failure cannot be found, remove the diskettes and call your TI distributor. Before calling, please be reasonably sure that the diskettes are at fault and not the TX990 Operating System or FS990 computer.

2.8 POWER BASIC SAMPLE PROGRAM

Once POWER BASIC has been initialized, the user may immediately enter the following sequence of commands and statements prevented on the following page to verify that POWER BASIC is executing correctly. Other sample programs which may be entered and executed are provided throughout this manual and in Appendix C.

When POWER BASIC begins execution it displays the following banner message. The user then enters the "SIZE" command to display the amount of RAM area "free" for user program storage. (The amount of free RAM given in the following examples is dependent on system configuration and the Operating System being used).

```
CONFIGURABLE POWER BASIC REV FS.1.4
*READY
SIZE
PRGM:  0 BYTES
VARS:  0 BYTES
FREE:  18850 BYTES
       (SIZE numbers will depend on system configuration)
```

The following program may then be entered:

```
 10   DIM A(4)
 20   $A(0)="THE NUMBER IS"
 30   INPUT "INPUT NUMBER", N
 40   IF FRA(N)<>0 THEN PRINT $A(0);N;:: GOTO 60
 50   GOSUB 100  ! EVEN OR ODD INTEGER
 60   PRINT ", ITS SQUARE IS";N*N;", AND ITS SQUARE ROOT IS";
 70   IF N<0 THEN PRINT " UNDEFINED.":: GOTO 30
 80   PRINT SQR(N);"."
 90   GOTO 30
100   IF INP(N/2)*2=N THEN PRINT $A(0);" EVEN";::RETURN
110   PRINT $A(0);" ODD";
120   RETURN
```

The user may then display the program size and list the program as follows:

```
SIZE
PRGM:  282 BYTES
VARS:  4   BYTES
FREE:  18564 BYTES
LIST
  10   DIM A(4)
  20   $A(0)="THE NUMBER IS"
  30   INPUT "INPUT NUMBER", N
  40   IF FRA(N)<>0 THEN PRINT $A(0);N;:: GOTO 60
  50   GOSUB 100  ! EVEN OR ODD INTEGER
  60   PRINT ", ITS SQUARE IS";N*N;", AND ITS SQUARE ROOT IS";
  70   IF N< 0 THEN PRINT " UNDEFINED.":: GOTO 30
  80   PRINT SQR(N);"."
  90   GOTO 30
 100   IF INP(N/2)*2=N THEN PRINT $A(0);" EVEN";::RETURN
 110   PRINT $A(0);" ODD";
 120   RETURN
```

The RUN command will execute this program. The program will request numeric user input by prompting with the question mark as follows:

```
RUN
 INPUT NUMBER? 17  (carriage return)
 THE NUMBER IS ODD, ITS SQUARE IS 289, ITS SQUARE ROOT IS 4.1231.
 INPUT NUMBER? -6 (carriage return)
 THE NUMBER IS EVEN, ITS SQUARE IS 36, ITS SQUARE ROOT IS UNDEFINED.
 INPUT NUMBER? 2.35 (carriage return)
 THE NUMBER IS 2.35, ITS SQUARE IS 5.5225, ITS SQUARE ROOT IS 1.532971.
 INPUT NUMBER? (escape key)
 STOP AT 30
```

The user may enter the SIZE command to display the program size and the variable storage used by the program.

```
SIZE
PRGM:  282 BYTES
VARS:  44  BYTES
FREE:  18524 BYTES
```

All variables and program storage area may then be cleared as shown by the following sequence:

```
NEW
CONFIGURABLE POWER BASIC REV FS.1.4
*READY
SIZE
PRGM:  0 BYTES
```

```
VARS:  0 BYTES
FREE:  18850 BYTES
```

The user should refer to the remainder of this manual for the detailed syntax and explanation of the POWER BASIC Language.

SECTION III

GENERAL PROGRAMMING INFORMATION

3.1 GENERAL

This section contains general programming information about the POWER BASIC language. General language features such as syntax, editing commands, and error listings will be presented.

3.2 BASIC LANGUAGE

The POWER BASIC language is composed of commands and statements. Commands are used to list, edit, save, load, execute, and debug the user's BASIC programs on the Host POWER BASIC Interpreter. No commands are supported in a Target (Configured) POWER BASIC Interpreter. Commands begin with the command name (or the first three letters of the command name in most cases) and are executed immediately upon entry. Statements in POWER BASIC programs are designed to perform a task or solve a problem. Statements begin with a line number and may be displayed and modified by using POWER BASIC commands. The user may abort the command or statement entered by: 1) NOT using the carriage return key at the end of the line, but backspacing and retyping the line; or 2) striking the ESCAPE key.

3.3 POWER BASIC PROGRAM

A POWER BASIC program consists of one or more lines, each uniquely identified by a line number in the range 0 to 32,767, and each containing at least one POWER BASIC statement of the form:

```
<line number><POWER BASIC statement>
 100  For I=1  To 10
```

More than one statement may appear on a single line by separating the statements with a double colon (::).

```
<line number><statement 1>::<statement 2>:: ...
 50  A=10::  B=5::  C=0
```

A POWER BASIC statement cannot be continued onto the next statement line. All POWER BASIC statements are terminated (entered) by entry of the carriage return key.

POWER BASIC will generate automatic line number prompts for the user

to facilitate simple program statement entry. Auto-line numbering is initiated via the NUMBER command as described in Section IV, paragraph 4.6, or by use of the line feed key to terminate statement entry.

Auto-line numbering using the line feed key is initialized to begin at statement number 10 and generates an increment value of 10 between subsequent statement numbers.

To initiate auto-line numbering using the line feed key, the user should either:

- Enter a line feed character as the first character of the line (to which POWER BASIC responds with line number 10), or
- Enter the first (starting) statement number and the associated statement and terminate the line with a line feed entry.

In both cases, the use of a line feed entry at the end of a statement (rather than the more commonly used carriage return) will result in line numbers being generated automatically in increments of 10 after each statement is terminated (entered). After the first statement has been entered via the line feed key, subsequent statements may be entered via either the line feed or carriage return keys and auto-line numbering will continue. To terminate auto-line numbering, enter an empty (vacuous) statement line with a carriage return.

POWER BASIC programs are executed beginning with the lowest numbered line and proceeding with the next highest numbered line until directed otherwise by a control statement, or until the last statement on the last line is executed. An example of a POWER BASIC program to compute the sum of the squares of two numbers is given below.

```
10 LET X=3
21 LET Y=4
33 LET Z=X*X+Y*Y
40 PRINT Z
57 STOP
```

The POWER BASIC line number is also used to associate program editing activities with a particular statement line in the program.

3.4 SOURCE STATEMENT FORMAT

3.4.1 LINE NUMBER FIELD

The line number field is the first field of any program line and is a decimal integer between 1 and 32,767, inclusive. This field, which starts in the first print position, must not contain any embedded

blanks and must be followed by at least one blank.

3.4.2 STATEMENT FIELD

The statement field follows the line number in a program line and contains one or more POWER BASIC statements separated by double colons (::). Each statement is comprised of a POWER BASIC keyword followed by a number of constants and/or variables separated by POWER BASIC operators. All keywords must be entered in upper case.

3.4.3 TAIL REMARK

The tail remark is separated from the statement field by an exclamation point (!) and can be used for source statement documentation. All characters following the exclamation point are treated as a remark and are not executed. Note that no tail remarks are configured into the final Target POWER BASIC Interpreter to conserve user program storage area in the final application.

3.4.4 CHARACTER SET

The character set for POWER BASIC is the upper and lower case alphabet A-Z; numbers 0-9; and special characters !"#$%&'([ ])*:=-@+;,.?/. Non-printable control characters may be specified by enclosing the hex representation of the character within angle brackets. For instance, a form feed, (ctrl)L, is specified by "<0C>"; a bell, (ctrl)G by "<07>". The phrase "(ctrl)" indicates the key code generated when the user holds down the control key while depressing the key corresponding to the character immediately following.

3.4.5 SPECIAL KEYBOARD CHARACTERS

The Host POWER BASIC Interpreter is designed to communicate, via the TX990 Operating System, to either of two terminal devices. The user must select and execute the appropriate POWER BASIC module which interfaces to the terminal device supported by the TX990 Operating System being used. The terminal devices supported by the two POWER BASIC modules are the 911 and 913 Video Display Terminals (VDT's). The key codes generated by these terminals differ slightly in their representation. Therefore, the POWER BASIC modules each contain a translate table to convert these codes to a standard code used by the POWER BASIC Package. Table 3-1 lists the keys of the VDT911 and VDT913 terminal devices which correspond to the special functions used by the POWER BASIC package. All other keys on the keyboard retain their normal functions.

The phrase "(ctrl)" indicates that the user holds down the control key while depressing the key corresponding to the character immediately following. For example: "(ctrl)H" means depress the "H" key while holding down the key marked "CTRL" or "CTL". The control (or editing) characters are not echoed on the terminal, nor are they stored in the input buffer. The keys F0, F1, F2, and F3 correspond to the function keys in the top row of the 911 or 913 Viceo Display Terminals. All illegal characters are echoed as a bell and otherwise ignored.

TABLE 3-1. SPECIAL FUNCTION KEYCODES

SPECIAL FUNCTIONS	INTERNAL CODE(HEX)	911 VDT	913 VDT
CARRIAGE RETURN (CR)	D	RETURN ENTER (ctrl)" M	NEW LINE
LINE FEED (LF)	A	(down arrow) "(ctrl)" J	(down arrow)
PERCENT (%)	25	%	F3
ESCAPE	1B	UNLABELED FUNCTION KEY*	RESET
BACKSPACES AND REMOVE CHARACTER	7F	DEL CHAR "(ctrl)" ___	DEL CHAR
DISPLAY LINE (ln) FOR EDITING	5	ln F1 ln"(ctrl)" E	ln F0
BACKSPACE CURSOR	8	(left arrow) "(ctrl)" H	(left arrow)
FORWARD SPACE CURSOR	12	(right arrow) "(ctrl)" F	(right arrow)
DELETE n CHARACTERS	4	F3 n "(ctrl)" D n	F2 n
INSERT n BLANKS	9	F2 n "(ctrl)" I n	F1 n
HOME CURSOR	6	HOME	HOME
CLEAR SCREEN AND HOME CURSOR	3	UNLABLED KEYPAD** KEY	CLEAR

* - Unlabeled function key in upper right hand corner of keyboard

** - Unlabeled key located in the upper right-hand corner of the keypad to the left of the keyboard

3.5 EDIT MODE COMMANDS

To aid in program writing and debugging, an advanced editor is contained in POWER BASIC. The editor uses the special control characters of Table 3-1 to perform these special editing features. To maintain consistency with Evaluation and Development POWER BASIC, the special function control keys (eg., "(ctrl)H") will be used in the following examples; however, the additional special characters of the 911 or 913 VDT's may be used to perform these editing functions.

All characters displayed on the terminal device are entered no matter where the cursor is located when a CARRIAGE RETURN or LINE FEED key is depressed.

The "(ctrl)E" feature allows editing of program lines previously entered into a POWER BASIC program. The form is:

```
(statement number) (ctrl)E
      100          (ctrl)E
```

The line will be displayed with the cursor remaining at the end of the line. Any editing as described below may then be performed.

The following examples illustrate the character insertion and deletion features of POWER BASIC. The cursor position is designated by an underline (_).

Entering "10(ctrl)E" results in:

```
10 A(J-1)=SQR(B(1)+B(1,2))_
```

Note that the second argument is missing from the first B array. Enter nine control H's to backspace to the offending location,

```
10 A(J-1)=SQR(B(1_+B(1,2))
```

and follow with (ctrl)I2. POWER BASIC will reply with

```
10 A(J-1)=SQR(B(1_ )+B(1,2))
```

after which the second argument can be entered and followed by a CARRIAGE RETURN to enter the edited line. If it is discovered later that a third argument of the square root is required, instead of retyping the line, enter:

```
10  (ctrl)E
```

and the computer will respond with:

```
10 A(J-1)=SQR (B(1,1)+B(1,2))_
```

Then enter one (ctrl)H followed by (ctrl)I7. The computer responds:

```
10 A(J-1)=SQR (B(1,1)+B(1,2)_        )
```

Enter the desired characters and press the CARRIAGE RETURN or LINE FEED key. The CARRIAGE RETURN enters line 10 into the program and returns to the keyboard mode, while the LINE FEED enters line 10 and prompts with the next sequential line number (line 20). The "(ctrl)Dn" operator is the reverse operation of the "(ctrl)In" operator. For example:

```
30 (ctrl)E
```

will display statement 30; which has an error.

```
30 REM CALCULATE SUB TOTATALS_
```

Entering 4 (ctrl)H's yields

```
30 REM CALCULATE SUB TOTATALS
```

Entering (ctrl)D2 yields

```
30 REM CALCULATE SUB TOTALS
```

which is the desired result. Complete the editing of this line by entering a CARRIAGE RETURN.

3.6. CONSTANTS

3.6.1 HEXADECIMAL INTEGER CONSTANTS

A hexadecimal integer constant is a decimal digit optionally followed by one to four hex digits followed by the letter H with no embedded blanks. A hex constant cannot begin with the letters A-F. In these cases they must begin with a zero. If more than four digits are given, only the right-most four digits are actually used. Valid combinations are OH to OFFFFH.

3.6.2 DECIMAL INTEGER CONSTANTS

A decimal integer constant is any integer between -32768 and 32767 inclusive.

3.6.3 DECIMAL REAL CONSTANTS

A decimal real constant is a numeric value with a decimal fraction. The number can have no more than 11 significant digits in the Host POWER BASIC and may not be larger than 10 or have a negative exponent less than 10. Real numbers may be expressed simply as a number followed by a decimal fraction, or may also have a exponent assumed to be a multiplier of 10 to that power. (Ex. 123.4 is equivalent to 1.234E2; 0.0000123 is the same as 1.23 E-5).

3.6.4 STRING CONSTANTS

A string constant is a string of characters enclosed within single or double quotes. Paired double quotes can be used to enclose single quotes, and the reverse is also true. (Example: 'THIS IS A STRING', "SO'S THIS", 'Demonstrates "quotes" within a string'. Non-printable characters may be included in string constants by enclosing their hex equivalent within angle brackets. (See Character Set, Paragraph 3.4.1). Actually, any character, printable or non-printable, may be included in a character constant. If you want both single and double quotes in a constant, single quotes could be represented as " 27 " or double quotes as " 22 ". POWER BASIC stores the constant exactly as it appears in the code, and interprets numbers between angle brackets only when printing them, or when reading them from a DATA statement (see Paragraphs 5.7 and 5.8). Angle brackets are NOT interpreted during assignment or comparison. Thus, the constant 'DON 27 T' will print as DON'T (five characters) but is kept as a string of eight characters. If a program requires the compact form for comparisons (i.e., looking for a specific combination of charcters in a source string), it is necessary to read the test string from a DATA statement or build it through concatenation of the individual characters.

For example:

```
$TST = 'DON' + %027H :: $TST = $TST + "T"
```

The above will place the desired five character string into the variable $TST. The % operator enables the POWER BASIC user to insert nonprintable ASCII character codes into string constants. The 7% operation inserts the decimal or hexadecimal ASCII code following the % symbol into the character string. For additional information refer to Section 5.8.2.

Numbers enclosed within angle brackets WILL be interpreted when printed. So if it is necessary to print out the statement "A<>B" (A is not equal to B), the angle brackets must be considered non-printable characters and specified as "A<3C><3E>B" (only the left bracket (<) is non-printable so that "A<3C>>B" is valid and will produce the same results).

3.7 VARIABLES

POWER BASIC supports four types of variables: simple numeric variables, numeric array variables, simple string variables, and string array variables. The two numeric variable formats are used extensively in POWER BASIC statements and arithmetic operations, while the two string variable formats are used extensively for string-character manipulation and output. Note that if any POWER BASIC numeric variable is referenced by a BASIC statement or command and the variable has not been previously defined, it will result in a "UNDEFINED VARIABLE" error. Also note that if any string variable is referenced and has not previously been defined, the string variable will be defined as a null string. A null string contains no characters except the null character (byte 00_{16}) as the first character of the string.

3.7.1 SIMPLE VARIABLES

Names for simple numeric variables must begin with a capital letter (A-Z) and may be followed by one or two more capital letters or a number in the range 0-127. Names for variables may not be the same as POWER BASIC key words or the beginnning of the same, i.e., SIN is not a valid name nor is LIS since it is the same as the first three characters in the command LIST.

Examples:

Valid names: A, ABC, CAT, A0, A123.

Invalid names: ABS (function name), A.B (non-alphanumeric), A130 (number out of range), AB1 (only 1 letter in letter-number combinations), 12B (first character must be letter), ABCD (too long).

3.7.2 NUMERIC ARRAY VARIABLES

The same rules given for formation of simple numeric variable names also apply to numeric array variables with the additional specification that numeric array variables must appear in a DIM statement. The DIM statement must be executed before the first reference to the variable from a simple variable of the same name; i.e., PRINT A and PRINT A(0) refer to two completely separate variables. When keying in a reference to an array variable, either parenthesis or square brackets may be used. (Both become square brackets internally and are subsequently printed as square brackets.)

3.7.3 SIMPLE STRING VARIABLES

Simple string variables follow the same rules given for simple numeric

variables with the added specification that the reference must be preceded by a dollar sign ($). Internally string data is stored left-justified and delimited by a null character (a zero byte). Characters are normally represented as 8-bit ASCII (normal 7-bit ASCII with the 8th bit set to zero). If the 8th bit is set to one, the interpreter will treat the character the same; however, a character with the 8th bit on is NOT equal to the same character with the 8th bit off! All strings are terminated by a null character. A simple variable in POWER BASIC is composed of 48 bits, or 6 eight-bit bytes. Thus, a maximum of five characters should be stored in a simple string variable terminated by a null. Longer strings should be stored in string arrays (dimensional string variables) as explained below. Any operation which attempts to place more than the maximum number of characters in a string variable will result data immediately following the string variable being overwritten.

3.7.4 STRING ARRAY

The same rules given for the formation of numeric array variables apply to string array variables with the added requirement that the name must be preceded by a dollar sign ($). The dollar sign, however, is omitted when defining array variables with the DIM statement. If the array is multi-dimensional, the data is stored internally with the right-most subscript varying most rapidly.

POWER BASIC, with 48-bit variables, stores 6 bytes maximum per array element. This is important if you wish to store a series of names longer than five characters in an array. For example, the array A is dimensioned by the statement, DIM A(2,1). The names "RHINOCEROS", "ELEPHANT", and "GIRAFFE" would be internally stored as:

```
$A(0,0)  :  RHINOC(EROS)  $A(0,1) : EROS
$A(1,0)  :  ELEPHA(NT)    $A(1,1) : NT
$A(2,0)  :  GIRAFF(E)     $A(2,1) : E
```

The data in the second column of the array is also output when printing $A(0,0),$A(1,0), or $A(2,0) since a string is delimited by a null character on printing. Since the string in the first column does not contain a null, BASIC continues on to the second column or until it finds a null. If that null is overwritten by placing something else there, unexpected results may occur. For example, by executing A(0,1) = A(2,0), and then printing $A(0,0), the result would be "RHINOCGIRAFFELEPHANT".

One additional characteristic of string array variables is that individual bytes in the variable may be referenced by specifying the byte index after the subscript. The first byte of a string is referenced by an index value of 1, and the index limit extends to the last character of the string. A semicolon is used to delimit the index from the subscript in this case. Example: $A(0,0;4) is "N" --

the fourth letter in RHINOS in the above example.

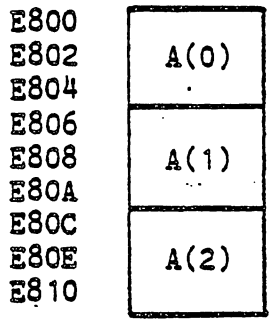
3.7.5 VARIABLE STORAGE

The following paragraphs will explain the internal variable storage structure used by POWER BASIC. This will be helpful when accessing variables of BASIC from a "CALLed" assembly language subroutine.

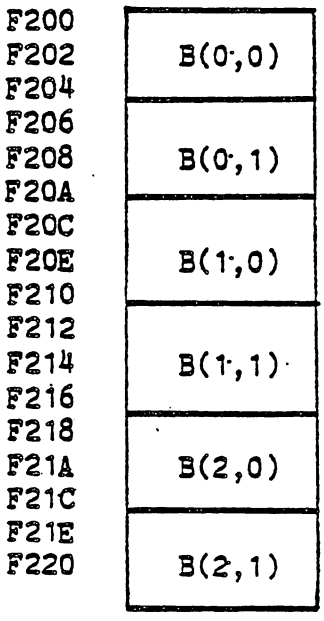
3.7.5.1 NUMBER ARRAY STORAGE

Arrays of numbers are stored in memory by row, with each element occupying 6 bytes. The storage of singly and doubly dimensioned arrays are illustrated in the two diagrams below. Larger dimensioned arrays are stored in a similar manner.

Single dimensioned array A with 3 elements starting at Hex address E800 :



Doubly dimensioned array B with 3 rows (first subscript) and 2 columns (second subscript) starting at hex address F200 :



As can be seen from the examples above, the address of an element in a singly dimensioned array is:

ARRAY BASE + 6 * (SUBSCRIPT)

eg. A(1) above would be:

E800 + 6 * 1 = E806

while the address of an element of a doubly dimensioned array element is:

ARRAY BASE + 6 * (MULTIPLER*SUBCRIPT1 + SUBSCRIPT2)

Where the multiplier is the maximum value of the second subscript + 1. For instance, B(1,0) above would be:

F200 + 6 * (2*1+0) = F20C

3.7.5.2 STRINGS AND STRING ARRAY STORAGE

Strings are stored one ASCII character per byte, and are terminated with a null byte. Recalling that POWER BASIC variables are 6 bytes in length, the examples below show the string storage format.

"BYE" stored in string variable $A at Hex address $F000_{16}$:

F000	"B"	"Y"
F002	"E"	00
F004	X	X

"BASIC" stored in starting at hex address $F020_{16}$:

F020	"B"	"A"
F022	"S"	"I"
F024	"C"	00

Strings may also be stored in dimensioned string variables, in which case each element has the same maximum length as a simple variable. The example below illustrates the storage of a string array $A having 3 elements and containing the string "POWER BASIC", starting at Hex address $EA00_{16}$.

$A(0)	EA00	"P"	"O"
	EA02	"W"	"E"
	EA04	"R"	"R"
$A(1)	EA06	"B"	"A"
	EA08	"S"	"I"
	EA0A	"C"	00
$A(2)	EA0C	X	X
	EA0E	X	X
	EA10	X	X

If the string of the above example were output using the "PRINT" statement, the following strings would result.

"PRINT $A(0), $A(1)" will result in:

POWER BASIC BASIC

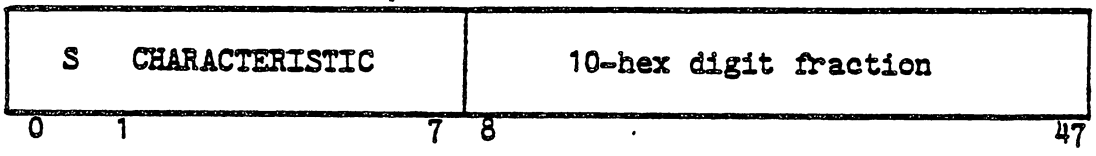
Strings may be empty or they may have any length up to their declared maximum. Care must be taken that strings of lengths larger than specified maximum are not placed into simple or dimensioned string variables, or other variables may be written over.

3.7.6 VARIABLE FORMAT AND ACCURACY

Any variable may contain an ASCII character string, a number, or both. Variable contents are completely program context dependent. Floating-point quantities in POWER BASIC are represented in 48 bits. The first bit in position 0 represents the sign of the number: 0 for positive numbers, 1 for negative numbers. The bits in positions 1-7 are the characteristic, or exponent, coded in excess-64 notation. The remaining bits of the floating-point number; positions 8-47, contain the mantissa or fractional portion of the floating-point number. The fraction is always recorded as a positive number; negative floating point numbers are <u>not</u> represented in complement form. The binary point of the fraction is understood to be just before bit position 8.

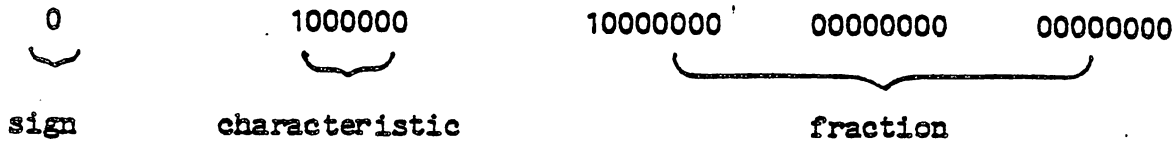
A floating-point number is represented by its fraction times a power of 16, with its sign attached to the result. The exponent indicating the power of 16 by which the fraction is multiplied is coded in the characteristic. The characteristic is 64 greater than the exponent. Excess-64 notation permits representation of a wide range of magnitudes, roughly from 16^{-64} to 16^{+63} (or 10^{-75} to 10^{+75}). The 48-bit POWER BASIC Interpreter provides approximately 11 digits of accuracy.

POWER BASIC Floating-Point Format:



Examples:

The pattern



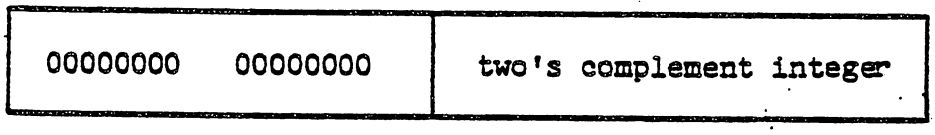
includes a characteristic of 64 (hex 40) and therefore an exponent of 0. The fraction is (binary) .1000...., or 2^{-1}, or decimal 0.50. Therefore, since the sign bit of 0 denotes a positive number, the number represented is $+0.5*16 = 0.50$.
The pattern

1	1000001	01010100	00000000	00000000
	C1	54	00	00

includes a characteristic of 65 and therefore an exponent of 1. The fraction is $.010101 = 2^{-2}+2^{-4}+2^{-6}$. The sign bit of 1 denotes a negative number, so the quantity represented is $-(2^{-2}+2^{-4}+2^{-6}) * 16^1 = -(2^2+2^0+2^2) = -(4+1+0.25) = -5.25$.

Integer quantities in POWER BASIC are represented in 32 bits, with zeros in bit positions 0 through 15 followed by the two's complement 16-bit integer in bit positions 16 through 31.

POWER BASIC Integer Format:



POWER BASIC will store a numeric quantity as either a floating point or integer value, dependent upon the magnitude of the quantity. If the number can be represented as a 16-bit two's complement integer, it will be stored in integer format; otherwise it will be stored in

integer format; otherwise it will be stored in floating-point format. Once a value becomes large enough and is converted into a floating point value, it is never converted back to integer format even if the number decreases to the two's complement integer range.

The following example program will demonstrate the internal integer and floating point formats used in POWER BASIC. The program requests user input by prompting with a question mark(?). The user may then enter any numeric value followed by a carriage return. The value will be displayed as a 32-bit integer with the first 16 bits zero, or as a 48-bit floating-point value.

```
10    INPUT A;
20    PRINT TAB(20)
30    FOR I=1  TO 48
40     IF I=2  THEN PRINT "  ";
50     IF I=9  THEN PRINT "  ";
60    PRINT #"0" BIT (A,I);
70    NEXT I
80    PRINT
90    GOTO 10

RUN
? 1.            0 1000001 00010000000000000000000000000000000000000000
? 1             0 0000000 00000000000000000000000100000000000000000
? 1.5           0 1000001 00011000000000000000000000000000000000000
? 123456789012  0 1001010 00011100101111110100110010001101000010100
? (ESC) Key
STOP AT 10
```

3.8 OPERATORS AND EXPRESSIONS

An expression is a list of variables and constants separated by operators. There are three types of POWER BASIC operators and expressions: arithmetic, logical, and relational.

3.8.1 ARITHMETIC OPERATORS

The following is a list of the valid arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
+	unary plus
-	unary minus

Note that POWER BASIC supports exponentiation to any floating quantity. Both positive and negative exponents are valid. Howe since POWER BASIC uses logarighms to calculate the exponentia only positive quantities may be raised to a given power. The e "LOG OF NON-POSITIVE NUMBER" will result if a negative quanti raised to any power.

For a given exponentiation when logarithms are used the resul answer may differ slightly from the anticipated results du internal floating point limitations.

ACTUAL	CONFIGURABLE POWER BASIC
2↑3 = 8	2↑3 = 7.9999999999 +1 - .00000000001

The approximately equal (= =) operator as presented is Section 3. may be used to overcome these floating point limitations. Anc result of using logarithms is that for a given exponentiation, resultant answer may differ slightly from the anticipated results to internal floating point limitations.

3.8.2 ARITHMETIC EXPRESSIONS

An arithmetic expression is any valid sequence of numbers, variabl operators, and parentheses (valid meaning that parentheses must properly balanced, and no two numbers or variables can be adjace and no two binary operators can be adjacent).

For example:

An expression may consist of a single operand:

```
8
SIN(A)
```

A sequence of operands may be combined by arithmetic operators:

```
X*Y
A*B-W/Z
```

Any expression may be enclosed in parentheses and considered to b basic operand:

```
(X+Y)/Z
(A+B)*(C-D)
```

Any expression may be preceded by a plus or minus sign:

```
+X
-(A+B)
-A+((TAN(-A))*2)
```

3.8.3 LOGICAL OPERATORS

The logical operators do bit wise operations on integers. They consist of the following:

LNOT	(unary)	1's complement of integer
LAND	(binary)	Bit wise AND of two integers
LOR	(binary)	Bit wise OR of two integers
LXOR	(binary)	Bit wise exclusive OR

3.8.4 LOGICAL EXPRESSIONS

Logical expressions are similar to arithmetic expressions. They both consist of variables, constants, parenthesis, and operators. The primary difference being that the operators are different for logical expressions. The logical operators perform a bit-wise logical operation on the operand(s).

For example, if A = 0AAAAH (hex "AAAA"), and B=05555H (hex "5555") and C = 0BBBBH, (hex "BBBB"), then

LNOT (A)	would equal	"5555"
A LAND B	would equal	0
A LOR B	would equal	'FFFF'
A LXOR C	would equal	'1111'

3.8.5 RELATIONAL OPERATORS

The relational operators are all binary operators that operate on two arithmetic expressions. They return values of 1 (TRUE) or 0 (FALSE). Relational operators consist of the following:

=	exactly equal
==	approx equal (plus or minus .0000007)
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal

3.8.6 BOOLEAN OPERATORS

The boolean operators are designed to work on the resultant TRUE or FALSE conditions set by the relational operators. However, they may also operate on variables within the program, in which case a zero value is considered False and a non-zero value variable is considered to be True. The boolean operators return values of 1 (True) or 0 (False).

Boolean operators consist of the following:

NOT	(UNARY)	Returns a TRUE value if expression evaluates to FALSE (non-zero); otherwise returns a FALSE value.
AND	(BINARY)	Returns a TRUE value if both expressions evaluate to TRUE (non-zero); otherwise returns a FALSE value.
OR	(BINARY)	Returns a TRUE value if either expression evaluates to TRUE (non-zero); otherwise returns a FALSE value.

3.8.7 BOOLEAN AND RELATIONAL EXPRESSIONS

Boolean and relational expressions are formed according to the following rules:

A Boolean or relational expression may consist of a single element:

```
NOT(A)
X<>3.14159
```

Single elements may be combined through the use of the Boolean operators AND and OR to form compound expressions such as:

```
A AND B
X OR Y
```

Any expression may be enclosed in parentheses and regarded as an element:

```
(T OR S) AND (R OR Q)
```

3.8.8 EXPRESSION EVALUATION

Expressions are evaluated left to right if the operators are of equal precedence, and there are no parentheses. If there are parentheses in the expression, the sub-expression within the innermost parentheses is evaluated first. Not all operators have equal precedence - operands

which are operated on by an operator of high precedence are evaluated before operations of low precedence.

The precedence of operators is:

1. Expressions in parentheses
2. Exponentiation and negation
3. *,/
4. +,-
5. <=,<>
6. >=,<
7. =,>
8. ==,LXOR
9. NOT,LNOT
10. AND,LAND
11. OR,LOR
12. (=) ASSIGNMENT

3.9 MULTIPLE STATEMENTS

A double colon (::) terminates a POWER BASIC statement and can therefore be followed by another statement on the same line. This saves memory, speeds execution and also allows for better program segmentation. A common divisor program using multiple statement lines is illustrated below:

The following examples demonstrate multiple statement lines. Note that this is not an executable program.

```
10    PRINT " A"," B"," C","GCD"
20    READ A,B,C
30    X=A:: Y=B:: GOSUB 200
40    X=G:: Y=C:: GOSUB 200
50    PRINT A,B,C,G:: GOTO 20
60    DATA 32, 384, 72
200    Q= INP (X/Y):: R=X-Q*Y
210    IF R=0 THEN G=Y :: RETURN
220   X=Y:: Y=R:: GOTO 200
```

All POWER BASIC statements may be preceded and followed by a double colon in multiple statement lines with the exception of the NEXT, DATA, and REM statements. The NEXT statement should not be preceded by another statement (i.e., should be the first statement of the line), the REM statement should not be followed by any statements on the same line, and the DATA statement should not be proceeded or followed by any statement on the same line.

3.10 KEYBOARD MODE

The Host POWER BASIC Interpreter executes statements in either

"execution" mode or "keyboard" mode. In keyboard mode, statement numbers are not entered, only one line is executed at a time, and control is returned to the user after the statement execution. This line may contain multiple statements properly separated by a double colon.

The system recognizes two kinds of input: statements and commands. (See Section 4 for Basic Commands and Section 5 for Basic Statements.) One and only one command may be executed per line with no statements on the line.

In execution mode, the program counter moves through the program executing statements. Execution mode is entered by RUN, CONTINUE, or GOTO; it returns to keyboard mode after any error, STOP, or when all statements have been executed, or when ESCAPE key is hit.

The following examples illustrate on the line calculations in keyboard mode. Note that ";" is equivalent to PRINT. The user must terminate each entry line with a carriage return and POWER BASIC will print the result. In the examples below all POWER BASIC responses are underlined for clarity.

```
PRINT 12*12; 144
;1/3;3^3; 0.3333333  27
;4*ATN1; 3.141593
;SIN(ATN1);SQR2; 0.7071068 1.414214
;EXP1;COS(4*ATN1); 2.71828  -1
;3*34/23^3-4+2 (3+5); 252.0084
I=1:: K=2:: PRINT I+K; 3
```

A FOR/NEXT loop can be executed in keyboard mode only if entered on one line; however, the loop cannot be ESCaped from.

The following types can only be executed in keyboard mode. They can only be entered one command per line and cannot be entered in a program:

CONTINUE	PURGE
LIST	RUN
LOAD	SAVE
NEW	SIZE
NUMBER	SOURCE
	STACK

3.11 ERRORS AND ERROR LISTING

The first run of new program may be free of errors and give the correct answers. But it is much more common that errors will be present and will have to be corrected. Errors are of two types:

errors of form (syntax, arithmetic, structure, or grammatical errors) which prevent the running of the program, and logical errors in the program which cause the computer to produce either the wrong answers or no answers.

Errors of form cause the error code and statement number in which the error occurred to be printed and program execution stops. Logical errors are often much harder to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. A line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the carriage return key. A line can be inserted only if the original line numbers are not consecutive numbers. For this reason, most programmers will start out using line numbers that are multiples of five or ten to leave space for the inevitable changes and corrections.

Corrections can be made at any time before or after a run. Since the computer sorts lines (and arranges them in order), a line may be retyped out of sequence. Simply retype the offending line with its original line number. If after examining a program the errors are not obvious and there are no grammatical errors, carefully select and insert temporary PRINT statements to see if the machine is computing what you wanted.

POWER BASIC displays error messages to indicate any errors which occur during program entry or execution.. POWER BASIC reports all errors using basically three formats.

The first format displays the error message and the statement number where the error occurred according to the following format:

```
***XXXXXXX*** AT YYYY
```

where:

XXXX is the error message
YYYY is the statement number

Examples:

```
***Syntax Error *** at 100

***Read out of data *** at 180
```

This error format is displayed whenever errors are encountered during program execution, and program execution will be terminated at the offending statement. The error format displays the statement line in which the error occurred. The offending statement line or other

segments of the program may then be edited to correct the reported error.

The second format displays only the error message when an error occurs. These type of errors are detected during keyboard mode statement execution, during statement or command entry, or during program LOADing from a device/file. They indicate that the most recently entered statement or command, or the most recently LOADed statement is in error. If the error is an error of syntax (i.e., something is wrong with the statement itself, typically a typing error, an omission, or an unrecognizable statement), the error is first output, followed on the next line by a repeat of the preceeding statement or command with the cursor positioned at the offending character. If a syntax error is detected during program LOADing, the error is output and the offending statement is output on the next line, but no cursor positioning is performed. LOADing then continues with the next statement on the device/file. If an error other than syntax occurs during command or keyboard statement execution, only the error is output. Any errors may then be corrected and the statement or command executed again.

The third format displays errors reported by the TX990 Operating System. These errors are reported to the user as hexadecimal codes in the format of

```
FILE I/O ERROR  0XXH
FILE I/O ERROR  0XXH AT YYYY
```

where

```
XX   is the error code
YYYY is the statement number
```

Examples:

```
File I/O Error 027H
File I/O Error 026H at 70
```

The following error codes and error messages may be issued by the POWER BASIC package. Refer to Appendix A-3 for the error codes and corresponding error messages reported by the TX990 Operating System.

CODE ERROR MESSAGE

```
 1 = SYNTAX ERROR
 2 = UNMATCHED DELIMITER
 3 = INVALID LINE NUMBER
 4 = ILLEGAL VARIABLE NAME
 5 = TOO MANY VARIABLES
 6 = ILLEGAL CHARACTER
 7 = EXPECTING OPERATOR
 8 = ILLEGAL VARIABLE NAME
```

```
 9 = ILLEGAL FUNCTION ARGUMENT
10 = STORAGE OVERFLOW
11 = STACK OVERFLOW
12 = STACK UNDERFLOW
13 = NO SUCH LINE NUMBER
14 = EXPECTING STRING VARIABLE
15 = INVALID SCREEN COMMAND
16 = EXPECTING STRING VARIABLE
17 = SUBSCRIPT OUT OF RANGE
18 = TOO FEW SUBSCRIPTS
19 = TOO MANY SUBSCRIPTS
20 = EXPECTING SIMPLE VARIABLE
21 = DIGITS OUT OF RANGE
22 = EXPECTING VARIABLE
23 = READ OUT OF DATA
24 = READ TYPE DIFFERS FROM DATA TYPE
25 = SQUARE ROOT OF NEGATIVE NUMBER
26 = LOG OF NON-POSITIVE NUMBER
27 = EXPRESSION TOO COMPLEX
28 = DIVISION BY ZERO
29 = FLOATING POINT OVERFLOW
30 = FIX ERROR
31 = FOR WITHOUT NEXT
32 = NEXT WITHOUT FOR
33 = EXP FUNCTION HAS INVALID ARGUMENT
34 = UNNORMALIZED NUMBER
35 = PARAMETER ERROR
36 = MISSING ASSIGNMENT OPERATOR
37 = ILLEGAL DELIMITER
38 = UNDEFINED FUNCTION
39 = UNDIMENSIONED VARIABLE
40 = UNDEFINED VARIABLE
41 = INVALID END-OF-USER RAM ADDRESS
43 = INVALID BAUD RATE
```

The following error messages result from the POWER BASIC/TX990 Operating System Interface:

```
50 = END-OF-FILE OCCURRED
51 = TABLE AREA FULL
52 = INVALID LUNO
53 = INVALID PATHNAME
54 = ZERO LENGTH RECORD
55 = INVALID FILE ACCESS
56 = POSITION ERROR
57 = INCOMPATABLE FILE TYPE
```

Note: Refer to Appendix A-3 for the additional and corresponding error messages reported by the TX990 Operating System.

SECTION IV

BASIC COMMANDS

4.1 GENERAL

POWER BASIC programs are created, executed, and debugged through interaction with the POWER BASIC system. The system recognizes two kinds of input: statements and commands. POWER BASIC commands direct and control system functions which include initiating computer operation, and storing and listing of programs. Commands cause immediate computer interaction thereby allowing operator control. Statements perform a sequentially assigned programmed task. Any command may be entered once BASIC has been initialized. An error message is generated when an improper or illegal entry is attempted.

Commands are used during program development and debug. For this reason, the Host POWER BASIC Interpreter will execute all of the commands presented in this section. None of the POWER BASIC commands may be entered into any application programs, and the Configurator will not accept any of the commands of this section. Therefore the Target POWER BASIC Interpreter and application may not contain any POWER BASIC commands.

4.2 CONTINUE COMMAND

Form:

CONtinue

The CONTINUE command transfers control to the next statement of the BASIC program after the occurrance of a break condition. (The RUN command always starts at the first line.)

When the RUN command is entered, program execution begins at the first line and continues until a break condition occurs. The CONTINUE command may be used to continue execution after a break.

The program will stop (or break) when the user enters the ESCape key during program execution, a STOP or END statement is encountered, or an error occurs within the program.

4.3 LIST COMMAND

Forms:

```
LIST
LIST <line number>
LIST  <-line number>
LIST <line number><-line number>
```

The LIST command lists all or any portion of the current program. Entering only the command forces the entire program to be listed. By entering a line number or range of the line numbers, specific portions of the program can be listed. Neither the starting or ending line numbers need to be an existing line number. POWER BASIC will begin listing at the first line number greater than or equal to the starting line number and terminate listing at the last line number less than or equal to the ending line number.

Example:

```
LIST
```

results in a listing of an entire program, while

```
LIST 100
```

lists all the lines from 100 through end of program, inclusive.

```
LIST -35
```

lists all the lines form the beginning of the program through line 35, inclusive.

```
LIST 90-120
```

lists all the lines from 90 through 120, inclusive.

4.4 LOAD COMMAND

The LOAD command will read a POWER BASIC program into memory which has been previously "SAVEd" onto the specified device/file.

Form:

LOAD {<string constant> / <string variable>}

The string constant or string variable specifies the pathname of the device/file from which the POWER BASIC program is to be read. Typical pathnames used in the LOAD command would be DSC:PROG1/DEV, DSC2:VALUE/CNT, CS1, of CS2. The actual pathnames of devices are defined during TX990 Operating System generation. The user should reference has particular system generation for the device names to be used in the formation of pathnames. Refer to Section 5, paragraph 5.15.1 of this manual for valid device/file pathname constructs.

The LOAD command automatically opens the specified device/file, followed by a rewind operation. It then loads the program and performs a close operation on the device/file.

As statements are read from the device/file, they are interpreted into internal pseudo-code and stored into the user program RAM area of POWER BASIC. If any invalid statements are read during LOADing, the corresponding error message, followed by the offending statement in which the error occurred will be output on the terminal device. Invalid statements typically have errors in form, such as syntax, structure, or grammatical errors. The offending statement is not stored into the user program RAM area of POWER BASIC, and the LOADing procedure will continue with the next statement on the device/file. Loading continues until the end-of-file is read on the the specified device/file. Control then returns to the keyboard of the terminal device.

When a POWER BASIC program is LOADed, only those statements with statement numbers loaded from the device/file will be inserted into the user program area. The statements already in memory having different statement numbers will not be affected. Any statements in memory which have the same numbers as the program on the device/file will be overwitten by the loaded program.

> Note that the user must insure that the file specified by the LOAD statement does contain a POWER BASIC program which has been properly SAVEd in the file. If a file which does not contain a POWER BASIC program is specified by the argument of the LOAD statement (eg., a file to which BINARY data has been written), unpredictable results may occur when this device/file is accessed for the LOAD.

Examples:

```
LOAD "DSC2:PROG1/SRC"     !LOAD BASIC PROGRAM FROM DISKETTE FILE
LOAD "CS1"                !LOAD BASIC PROGRAM FROM CASSETTE #1
LOAD "DSC:CONTROL/PG1"    !LOAD BASIC PROGRAM FROM DISKETTE FILE
```

4.5 NEW COMMAND

Forms:

```
NEW
NEW <address>
```

The NEW command without an address deletes the current user program and clears all variable space, pointers, and stacks. POWER BASIC responds with "*READY" and awaits the entry of new BASIC programs. The programs may be retrieved later if they have been SAVED.

The form of the NEW command with an address parameter is used by the Host POWER BASIC to limit the amount of RAM which can be used by the POWER BASIC system for user program area. When POWER BASIC is executed, it automatically sizes and clears the system RAM area starting from the top of the POWER BASIC Interpreter and checking sequential memory locations up through memory until a write/read mismatch is detected. All of the detected RAM area is then allocated to POWER BASIC to be used for interpreter overhead and user program area. Under some circumstances, it may not be acceptable for POWER BASIC to use all available contiguous RAM. In some applications it may be required to reserve a specific block of RAM area for use by the application. For example, a small area of RAM is required for any assembly language subroutines which are to be CALLed by the POWER BASIC program. For this reason, the NEW command with the address parameter was introduced. The upper bound of RAM memory is set to the specified address, and all POWER BASIC pointers are initialized to correspond to this new memory configuration. The Host POWER BASIC Interpreter will not use RAM above this address. Therefore an area of RAM can be reserved for application use from the specified address parameter on up toward memory address $F7FF_{16}$. POWER BASIC verifies that the specified address is not less than the absolute minimum valid address (that is, it will not permit the user to "walk over" POWER BASIC). This address denotes the end of the POWER BASIC Interpreter and will vary between operating systems. If the specified address is less than this value, an INVALID END-OF-USER RAM error will result. However, no address checking is performed to verify that the specified address is a valid RAM address. Care must be taken when specifying the address to make sure that RAM does acutally exist at that address and is contiguous from low memory (0000_{16}) up to that address. Eratic operation of POWER BASIC will result if an invalid RAM address is

specified. The user must also be sure that the software which uses the free area does not overlap past the specified address into the workspace area of POWER BASIC. In addition to setting the UPPER memory bound, the NEW command also deletes the current user program, and clears all variable space, pointers, and stacks.

Examples:

```
NEW

NEW 0E800H
```

4.6 NUMBER COMMAND

Form:

```
NUMBER
NUMBER <start line number>
NUMBER <start line number>[,increment]
```

The NUMBER command enables the automatic line number prompt. This prompt is terminated by a null line entry. The simplest form of the command (the command word itself) assigns 100 as the starting line number and prompts the line numbers in increments of 10.

A different starting line number can be specified as a parameter of the command. Thus, the command

```
NUMBER 1000
```

prompts the line numbers in increments of 10 starting at line 1000.

The command

```
NUMBER 25,5
```

prompts the line numbers in increments of 5 starting at line number 25.

Entering a null line by entry of the carriage return key immediately following the line number prompt will terminate automatic line number prompting.

Automatic line numbering with fixed start and increment values may also be initiated via the line-feed entry key as explained in Section 3, paragraph 3.3.3.

4.7 PURGE COMMAND

The PURGE command allows the user to delete specified segments of the

current program.

Form:

```
<line number> PURGE <N1> TO <N2>
              PURGE <N1> TO <N2>
```

where N1 and N2 are the start and end line numbers, respectively.

The PURGE statement is designed to delete blocks of consecutive BASIC statements. PURGE will delete all statements from N1 to N2 inclusive. POWER BASIC will begin deleting statements at the first statement greater than or equal to N1 and will continue to the last statement less than or equal to N2.

Examples:

```
LIST
 10   REM SUBROUTINE TO PRINT I AND I*I
 20   FOR I=1 TO 10
 30   PRINT I, I*I
 40   NEXT I
 50   STOP

PURGE 15 TO 45
*READY

LIST
 10   REM SUBROUTINE TO PRINT I AND I*I
 50   STOP
```

4.8 RUN COMMAND

Form:

```
RUN
```

The RUN command clears all variable space, pointers, and stacks and directs the system to begin execution of the current BASIC program at the lowest line number. The command

```
RUN
```

will execute the user's POWER BASIC program currently in RAM.

4.9 SAVE COMMAND

The SAVE command will write the users POWER BASIC program currently in

Form:

```
SAVE <string constant>
     <string variable>
```

The string constant or string variable specifies the pathname of the device/file to which the POWER BASIC program currently in memory will be written. Typical pathnames used in the SAVE command would be DSC:PROCESS/CNT, DSC2:MOTOR/PRG, CS1, or CS2. The actual pathnames of the devices are defined during TX990 Operating System generation. The user should reference his particular system generation for the device names to be used in the formation of pathnames. Refer to Section 5, paragraph 5.15.1 for valid device/file pathname constructs.

The SAVE command automatically opens the specified device/file, followed by the rewind operation. It then saves the program and performs a close on the device/file.

If the program is SAVEd to a diskette file, the file must be created by the BDEFS or BDERF statements before the SAVE command is executed. Typically, BASIC programs are SAVEd into sequential files, since these files are only used by the SAVE and LOAD commands, and otherwise are not accessed by the user. Attempts to SAVE a program to a diskette file which does not exist will result in a TX990 undefined file error.

The program is output in source form, that is, in POWER BASIC statement format; in contrast to the pseudo-code form in which it is internally stored. Since SAVE performs in this manner, the user may enter a SAVE "LP" command and the program will be "listed" to the line printer device.

Examples:

```
BDEFS "DSC2:PROC/SRC"   !DEFINE FILE FOR SAVE (See Section 5.15.2)
SAVE "DSC2:PROC/SRC"    !SAVE POWER BASIC PROGRAM TO DISKETTE FILE
SAVE "CS1"              !SAVE POWER BASIC PROGRAM TO CASSETTE #1
SAVE "LP"               !SAVE (LIST) POWER BASIC PROGRAM TO LINE
                         PRINTER
```

4.10 SOURCE COMMAND

Form:

```
SOURCE
```

The SOURCE command prints the number of source bytes in the currently loaded user's program. This indicates the number of bytes that would be written by the SAVE command.

Example:

```
SOURCE
SRCE = 243 BYTES
```

4.11 STACK COMMAND

Form:

```
STACK
```

The STACK command lists the user GOSUB stack beginning with the first GOSUB return statement number pushed on the stack. These numbers indicate the return statement number for nested GOSUB's in the user's program. This command is helpful in the debugging of an application program.

Example:

```
 10  GOSUB 100
 20  A=SQR(I)
 99  STOP
100  REM
110  GOSUB 200
120  I=5
190  RETURN
200  REM
210  REM
280  STOP
290  RETURN

RUN
STOP AT 280
STACK
#20
#120
*READY
```

The STACK command will point back to a calling line that contains multiple statements to indicate that execution will continue with the statement immediately following the GOSUB. The following example illustrates this.

Example:

```
 10  GOSUB 100
 20  A=SQR(I)
 99  STOP
100  REM
```

```
110   GOSUB 200 :: J=5*5
120   I=5
190   RETURN
200   REM
210   REM
280   STOP
290   RETURN

RUN
STOP AT 280
STACK
#20
#110
*READY
```

4.12 SIZE COMMAND

Form:

```
SIZE
```

The SIZE command monitors memory usage by listing the current program size, variable space allocated, and the free memory in bytes.

Example:

```
SIZE
PRGM:  0 BYTES
VARS:  0 BYTES
FREE:  18850 BYTES
```

SECTION V
BASIC STATEMENTS

5.1 GENERAL

This section discusses the POWER BASIC program statements. Statement formats are presented and their uses are described.

During BASIC program execution, control may pass to any statement. Some statements have no effect on the program when encountered and are called nonexecutable; all others are called executable.

Statements form the basis of all functional POWER BASIC programs. Each statement of a BASIC program may occupy only one line; however, numerous statements may appear on each line when delimited by a pair of colons (::).

BASIC statements are divided into the following categories:

- Remarks
- Dimension Declarations and Specifiers
- Function Definition
- Assignment
- Control
- Input/Output
- Interrupt Processing
- CRU Base Assignment
- Time of Day
- Randomize Number Seed
- Program Escape/Noesc
- External Subroutine
- File Management

Table 5-1 briefly describes each statement.

TABLE 5-1

POWER BASIC STATEMENTS

STATEMENT	FUNCTION	USE
REM	Comment Line	Program documentation/explanation
DIM	Size Specifier	Dimensions strings, vectors, and matrices
DEF	Function Definition	Defines a statement function
LET	Assignment	Evaluates expressions and assigns value
EQUATE	Assignment	Two symbols refer to the same variable
GOTO	Control	Transfers unconditionally
IF	Control	Conditionally executes statement(s) on TRUE condition
ELSE	Control	Conditionally executes statement(s) on FALSE condition
GOSUB	Control	Transfers to BASIC subroutine
RETURN	Control	Returns from BASIC subroutine
POP	Control	Removes top return address from GOSUB stack
ON	Control	Computed GOTO or GOSUB
FOR	Control	Defines top of loop and loop parameters
NEXT	Control	Delineates loop scope
ERROR	Control	Transfers on error condition
STOP	Control	Stops program
END	Control	Stops program
BYE	Control	Terminates POWER BASIC and returns to TX990 Operating System

TABLE 5-1 (cont)

BASIC PROGRAM STATEMENTS

STATEMENT	FUNCTION	USES
READ	Internal Input	Reads from internal data block
DATA	Internal Input	Defines internal data block
RESTOR	Internal Input	Resets internal READ to first data block element
INPUT	I/O	Reads from terminal
PRINT	I/O	Prints on output device
TAB	I/O	Formats output into columns
DIGITS	I/O	Specifies number of significant digits output
SPOOL	I/O	Assigns unit number to specified file/device
UNIT	I/O	Designates print output device
BAUD	I/O	Designates baud rate of I/O device
IMASK	Interrupt Processing	Sets interrupt mask
TRAP	Interrupt Processing	Assigns interrupt level to interrupt subroutine
IRTN	Interrupt Processing	Returns from interrupt subroutine
BASE	CRU base assignment	Sets the CRU base address
TIME	Time of day	Sets, displays, or stores the 24-hour time of day clock
RANDOM	Set Random seed	Sets the seed of the pseudo random number generator
ESCAPE/ NOESC	Program escape/ no escape	Enables or disables the escape key to interrupt program execution
CALL	External Subroutine	Transfers to external subroutine

TABLE 5-1 (cont)

BASIC PROGRAM STATEMENTS

STATEMENT	FUNCTION	USES
BDEL	File Management	Deletes specified file
BCLOSE	File Management	Closes specified file
RESET	File Management	Closes all open files
COPY	File Management	Copies one file to another
BDEFR	File Management	Defines a relative record disc file
BDEFS	File Management	Defines a sequential disc file
BOPEN	File Management	Opens specified file for sequential or relative record access depending on file type
BINARY 1	File Management	Sets BINARY luno and specifies the number of bytes to be read or written for any subsequent BINARY commands.
BINARY 2	File Management	Writes an assigned number of bytes from each expression to the BINARY luno.
BINARY 3	File Management	Enables the user to read an assigned number of bytes into each variable from the BINARY luno.
BINARY 4	File Management	Allows access to a given byte position within a relative record disc file.

5.2 COMMENT OR REMARK (REM) STATEMENT

Form:

[line number] REM text

The REM statement is used to insert remarks (comments) in a program. REM may contain any textual information. It has no effect when encountered in execution; however, its line number may be used as the argument of a GOTO or GOSUB statement. Tail remarks may also be inserted into a program by separating the remark field from the statement field by an exclamation point (!). For additional information on tail remarks, refer to Section 3.4.3.

Examples:

```
10   REM THIS IS A COMMENT
100  REM CHECK FOR X=0
```

Target POWER BASIC Interpreter: Note that no REM statements or tail remarks are Configured into the final Target POWER BASIC Interpreter to conserve user program storage area required in the final application.

5.3 DIMENSION STATEMENT

Dimension declarations are used to specify the size attributes for subscripted variables within the program.

Form:

[line number] DIM <var(dim[,dim]...)>[,....]

The DIM statement dynamically allocates user variable space for array variables. Dimensioned (array) variables must be declared by the DIM statement before the variables are used. Once dimensioned, attempts to redimension an array variable to a larger array size will result in an error message and attempts to redimension to a smaller size will be disregarded.

Array sizes are specified by indicating the maximum subscript values in parentheses following the array name. Subscripts of dimensioned variables may be any numeric quantity including constants, simple variables, other dimensioned variables, or even function calls. If a floating point value is returned for the subscript value, only the integer portion will be used in the dimension statemement. The number of dimensions and the dimension size for the array declaration is limited only by the user's available memory. An error will occur if the dimensioned variable requires more variable space than is currently available in the user's partition. Dimensioned variables always use the 0 subscript as the first element in the array.

Examples:

```
 10   DIM  A(10),B(10,20)
100   DIM  A1(10),B1(20,30),B15(10,10,10)
      DIM  CAT(C,D),DOG(SQR(N),3,F)
```

The first statement allows for the entry of an array of 11 elements (0-10) into A, and of an array of 11 x 21 elements into the two dimensional array, B. The two remaining statements dimension arrays in a similar manner.

String variables must be dimensioned as numeric variables, e.g., $A must be dimensioned as A(10) not $A(10). Thereafter, the dimensioned numeric variable may be referenced as a string variable by preceding the variable with a dollar sign ($). The string array A dimensioned above should be referenced as $A(0) through $A(10).

Examples:

```
20   DIM  CAT(10),DOG(8)
```

This statement defines CAT to be a one dimensional array with 11 elements and defines DOG as a one dimensional array of 9 elements. Hereafter, these arrays may be considered as string arrays by referencing the variables via $CAT(0) through $CAT(10) and $DOG(0) through $DOG(8).

Strings are stored one character per byte with a null character used to terminate the string. Hence, simple string variables and single array elements which are 6 bytes in length can contain up to five characters. Dimensioned string variables can contain up to the number of elements times 6 minus 1 characters in Configurable BASIC, therefore, the dimensioned string variable $CAT can contain up to 65 characters.

5.4 FUNCTION DEFINITION

The DEF statement defines a user function. The defined functions are executed only when the function is referenced.

Forms:

[line number] DEF FN <letter>= <expression>

[line number] DEF FN <letter>(parm1 ,parm2 ,parm3) = <expression>

where:

parameters are single alphabetic letter dummy variables
expression is any valid POWER BASIC expression.

The DEF statement may appear anywhere within a BASIC program and the defined functions may be used in any expression. That is, once defined, the functions may be used in the same way as the built-in mathematical functions explained in Section 7. When the function is referenced, the expression is evaluated and the parameters, if any, are replaced by the arguments given in the reference. Within the expression the parameters may appear only as numeric variables. The user may define functions using up to three dummy parameters. All (dummy) parameters may only be single character variables in the function definition. However, when calling the function the user may use any valid POWER BASIC variable (either simple or dimensioned) to replace the dummy variables of the called function.

The expression may include any combination of intrinsic functions, other user-defined functions, or may involve any other variables in addition to the ones used in the argument of the calling function. Parameter names are dummy (local) variables of the defined function, and have no meaning outside of the function definition.

The use of the DEF statement is limited to those functions whose expression may be evaluated within a single BASIC statement.

The name of the defined function must be three letters, the first two of which must be FN followed by a single letter; e.g., functions FNA through FNZ may be defined by the user. The same letter which defined the function may also be used as a parameter of the function as shown below.

The following examples illustrate the various forms of the DEF statement. Note that this is not an executable POWER BASIC program.

```
20   DEF  FNA(X,Y)=X/Y+5
30   DEF  FNB = A/B + C-15
40   DEF  FNC(I,J) = I*K/J + FNB - FNA(I,J)
50   DEF  FND(N) = N*N/2
60   DEF  FNI(I,J) = I*J/SQR(I)
```

5.5 VARIABLE ASSIGNMENT

5.5.1 LET STATEMENT

The LET statement assigns a value to a variable where the variable is set equal to an expression consisting of variables and/or constants separated by operators. The variable being evaluated may appear within the expression. The newly calculated value of the variable replaces the old value.

In POWER BASIC the letters LET may be omitted from the statement so only an equation appears. The LET statement may have either of the following forms:

[line number] LET <variable> = <expression>
[line number] <variable> = <expression>

where

variable is a string variable, numeric scalar variable, or array element.

The assignment statement assigns an expression value to a variable. The variable and the expression must both be either string or numeric. The following examples illustrate the assignment statement. Note that this is not a meaningful POWER BASIC program.

```
        A=5
        B=10
        LET C=A+B
10      LET  X=1
20      LET $A(2)=$C+"NOW"
30      LET Q2(L)=Q2(L+1)+3
40      LET H=6
50      D=5
60      F=A/B+3
100     LET Z(I,J) = 3*X-4*Y
120     $AB="STOP"
```

5.5.2 EQUATE STATEMENT

The EQUATE statement enables a simple variable to be equated to another simple variable or to an element within an array.

Form:

[line number] EQUATE<simple variable>,<variable>;<simple variable>,<variable>...

The EQUATE statement may appear anywhere within the program. The simple numeric variable is equated in the symbol table to the associated numeric variable and thereafter either may be used when referencing that variable. This is a valuable technique for passing parameters through variables to a common subroutine. The subroutine may access its parameters from specifid simple variables (e.g., A, B, and C), and the calling routine may assign these simple variables via an EQUATE prior to execution of the GOSUB statement. The user may equate any other simple variable or array element to these simple variables.

Simple variables may also be equated to specific bytes within a dimensioned variable, (e.g., EQUATE TAG, FLAG [0;3]). These bytes may then be accessed by specific names and functions as byte flags within the application program.

In applications where memory usage is critical, the user should note that simple variables (e.g., A,$B) require less storage allocation when referenced in a program than dimensioned variables (e.g., A(1),$B(1)). Therefore, in applications where dimensioned variables are frequently referenced, memory area may be conserved by using the EQUATE statement to assign dimensioned variables to simple variables.

The following examples illustrate the use of the EQUATE statement. A carriage return is represented by (CR), and all user responses are underlined.

Examples:

```
10    DIM A(10)
20    EQUATE  NAM,A(0);PHN,A(5);SSN,A(7)
30    INPUT   $A(0),$A(5),$A(7)
40    PRINT   $NAM,$PHN,$SSN
50    STOP

RUN
 : JOHN DOE (cr) : 492-1356  (cr) : 486-57-4392 (cr)
JOHN DOE          492-1356        486-57-4392
STOP AT 40
```

Additional examples:

```
10    EQUATE I1,I(1);T23, T(2,3); F2, FLG(0;2)
20    EQUATE I1,PI;JJ,P2
30    GOSUB 1000

1000  I1 = I1*ATN(1)
1090  RETURN
```

Target POWER BASIC Interpreter: The EQUATE statement is not supported by the POWER BASIC Configurator, therefore the EQUATE statement should not be present in a final application program which is to be configured into a customized (Target) POWER BASIC Interpreter.

5.6 CONTROL AND COMPUTED TRANSFER STATEMENTS

BASIC statements are executed sequentially unless altered by control statements. Control may be accomplished by an unconditional branch, subroutine branch, computed branch, or loop.

5.6.1 UNCONDITIONAL GOTO STATEMENT

When the computer encounters a GOTO statement, it jumps to the program line number specified in the statement. The program executes the statement at the speceified line number and continues in sequence with the statements that follow.

Form:

[line number] GOTO <line number>

The "GOTO" statement must be entered without any embedded blanks. If the GOTO statement is not preceded by a line number; program execution begins at the line number specified immediately after the GOTO statement.

Examples:

```
     GOTO  200    Begins execution at statement 200
100  GOTO  140    Transfers control to statement 140
```

The following program illustrates the GOTO statement:

```
20  INPUT A
30  GOTO 50
40  STOP
50  PRINT A
60  GOTO 40
```

The program execution sequence is line numbers 20, 30, 50, 60 and 40 where execution stops.

5.6.2 CONDITIONAL IF-THEN-ELSE STATEMENT .

The IF-THEN-ELSE statements provide capability for conditional execution of program statements.

5.6.2.1 IF-THEN STATEMENT

The IF statement alters sequential execution of the program depending on the state of the specified condition.

Forms:

```
[line number] IF <expression> THEN BASIC<statement(s)>
[line number] IF <expression> relation <expression> THEN <BASIC statement(s)>
[line number] IF <string><relation><string> THEN<BASIC statement(s)>
[line number] IF <string> THEN<BASIC statement(s)>
[line number] IF <string><relation><string><,expression>THEN<BASIC statement(s)>
```

The condition may be any variable, numeric expression, relational expression, logical expression, string variable, string relational expression, or function which can evaluate to a zero or non-zero value. Two expressions or strings are compared according to the given relation and a true or false condition results. If the second string is followed by a comma, the expression following the comma indicates the number of characters to be compared. If only a single expression or string is given, the condition is considered false if the expression is zero or the string is null; otherwise, it is considered true.

If the condition is true, the statement(s) following the THEN clause on the same line will be executed. If the condition is false, the statement on the line following the IF-THEN statement will be the next statement executed. Any POWER BASIC statement or statements (including GOTO's and other IF-THEN statements) may immediately follow the THEN clause. They cannot extend to the next statement line because statement execution continues at the next statement line when a false condition occurs. The IF and THEN clauses must appear on the same statement line.

Examples:

```
20  IF A=0 THEN GOTO 100
30  IF SQR(J) =4 THEN K=J*J/I::PRINT J,K
40  IF I+2 THEN PRINT I
50  IF $A=$B THEN PRINT $A
60  IF $A THEN $B=$A
70  IF CRU(11) THEN CRU(12)= 1::GOTO 200
80  IF $A=$B,3 THEN GOTO 200     (compares first three characters
                                   of $A and $B)
```

5.6.2.2 ELSE STATEMENT

The ELSE statement enables conditional execution of POWER BASIC statements depending upon the true or false condition of the last executed IF statement.

Form:

[line number] ELSE <POWER BASIC statement>

IF-THEN statements set the ELSE flag to indicate the true or false condition of the last executed IF-THEN statement. Subsequent ELSE statements use the ELSE flag to determine whether the statement(s) following the ELSE are to be executed. When the IF condition is true, the THEN clause will be executed and all subsequent ELSE statements will not be executed. When the IF condition is false, the THEN clause will not be executed and all subsequent ELSE statements will be

executed. The ELSE statement must not be placed on the same statement line as the preceding IF-THEN statement because when the IF condition is false, no further statements on the IF-THEN line will be executed, and execution will continue with the next statement line. The ELSE flag remains set to the true or false condition until the next IF-THEN statement is executed at which time the flag is cleared and set to the new true or false condition. Several ELSE statements may appear between each IF-THEN statement, and each of these will be executed when they are encountered if the last executed IF-THEN statement resulted in a false condition. If a true condition resulted, each of these statements will be skipped. An ELSE statement always uses the last IF statement executed as its reference regardless of where it physically lies within the POWER BASIC Program. This enables blocks of statements to be conditionally executed or skipped.

Example:

The following program computes the function and prints the result:

Statement of function:

```
for X<1,  f=ABS(X),
for 1< =X < 2,  f=SQR(X),
for 2< =X,  f=ABS(X)-SQR(X)
```

Program solution:

```
10  IF X<1 THEN F=ABS(X)
20  ELSE IF X<2 THEN F=SQR(X)
30  ELSE F=ABS(X)-SQR(X)
40  PRINT X,F
```

5.6.3 SUBROUTINE (GOSUB, POP, AND RETURN) STATEMENTS

BASIC programs may contain internal BASIC subroutines. An internal subroutine is a sequence of BASIC statements performing a well-defined function or operation within the POWER BASIC program. Three types of statements govern access to a subroutine: a GOSUB statement for entry into the subroutine, a POP statement for exiting nested subroutines, and a RETURN statement for return to the calling program.

Forms:

```
[line number] GOSUB <line number>
[line number] POP
[line number] RETURN
```

An internal POWER BASIC subroutine may be invoked from any point within the program by using a GOSUB statement which specifies the entry point of the subroutine as a line number. Execution of the GOSUB statment pushes the address of the statement immediately following the GOSUB statement onto the GOSUB stack for return and passes execution to the specified line number.

A RETURN statement placed in the subroutine is an exit point from the internal POWER BASIC subroutine. A RETURN statement should be placed at each logical end of all subroutines. The RETURN statement causes execution to resume at the first statement following the GOSUB statement that transferred to the subroutine. During this transfer, the top return address is removed from the GOSUB stack. All subroutines should be exited only via a RETURN statement so the top return address will always be removed from the GOSUB stack. Unpredictable results occur if a subroutine is exited in any other fashion.

In Figure 5-1 GOSUB 90 involves statements on line numbers 90 (start of subroutine), 100, and 110 (end of subroutine). If a GOSUB statement is used, the subroutine it branches to must contain at least one RETURN statement. The example illustrates the simplest use of GOSUB and RETURN. The arrows indicate the flow of control in the program.

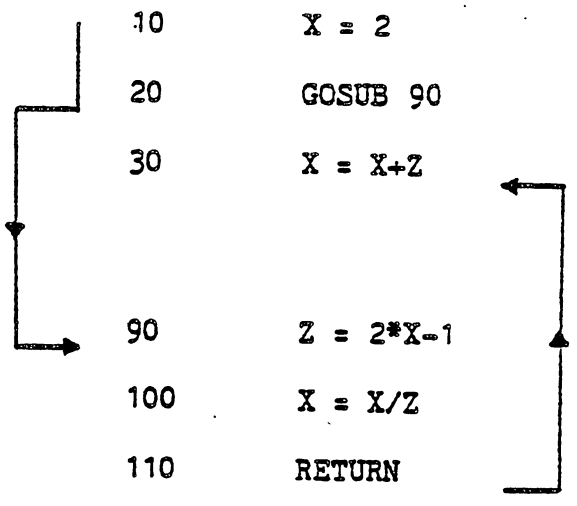


FIGURE 5-1

GOSUB Example

Subroutines may be nested by a subroutine containing a call to another subroutine; the inner subroutine is called a nested subroutine. Subroutines may be nested up to 20 levels in Configurable POWER BASIC.

A return address (first line number after the GOSUB) must be stored for each GOSUB statement until that statement is executed. The program in the following example contains nested subroutines and shows the actual execution sequence. Each GOSUB to a subroutine must be accompanied by at least one RETURN statement per exit path. The nested program and execution sequence of the example demonstrate entry to and exit from a subroutine.

```
LIST
 10   PRINT "ROOTS OF QUADRATIC EQUATIONS"
 20   PRINT
 30   REM - ENTER COEFFICIENTS A,B,C  OF A*X*X +B*X+C
 40   INPUT "COEFFICIENTS A= ";A;"   B= ";B;"  C= ";C
 50   GOSUB 100
 60   REM - RESTART OR END PROGRAM?
 70   INPUT "MORE DATA (1=YES, 0=NO)? "%1;N
 80   IF N<> THEN GOTO 20
 90   STOP

 100  REM - CALCULATE S=B*B-4*A*C
 110  S=B^2-4*A*C
 120  REM - COMPLEX ROOTS?
 130  IF S<0 THEN GOSUB 200   !COMPLEX ROOTS
 140    ELSE GOSUB 300        !REAL ROOTS
 150  PRINT                   !OUTPUT BLANK LINE
 160  RETURN

 200  REM - CALCULATE COMPLEX ROOTS
 210  Q=SQR ABS (S)
 220  R1=-B/(2*A)             !REAL PART
 230  R2=Q/(2*A)              !IMAGINARY PART
 240  PRINT "ROOTS (COMPLEX):  ";R1;" + OR -";R2;" I"
 250  RETURN

 300  REM - CALCULATE REAL ROOTS
 310  IF S=0 THEN Q=0
 320  ELSE Q=SQR (S)
 330  R1=(-B-Q)/(2*A)         !ROOT 1
 340  R2=(-B+Q)/(2*A)         !ROOT 2
 350  PRINT "ROOTS (REAL):  ";R1;",   ";R2
 360  RETURN
```

would produce the following results:

```
RUN
ROOTS OF QUADRATIC EQUATIONS

COEEFICIENTS A= 2  B= 1  C= -1
ROOTS (REAL):     -1,     0.5

MORE DATA (1=YES,  0=NO)?  1

COEFFICIENTS A= 1  B= 4  C= 6
ROOTS (COMPLEX):   -2 + OR - 1.414214 I

MORE DATA (1=YES, 0=NO)? 0

STOP AT 90
```

The following example shows the execution sequence of the ab
example. Note that all returns are performed via RETURN statements

Execution sequence:

```
 10   PRINT "ROOTS OF QUADRATIC EQUATIONS"
 20   PRINT
 30   REM - ENTER COEFFICIENTS A,B,C  OF A*X*X +B*X+C
 40   INPUT "COEFFICIENTS A= ";A;"  B= ";B;"  C= ";C
 50   GOSUB 100

100   REM - CALCULATE  S= B*B-4*A*C
110   S= B^2-4*A*C
120   REM - COMPLEX ROOTS?
130   IF S<0 THEN GOSUB 200    !COMPLEX ROOTS
140      ELSE GOSUB 300        !REAL ROOTS

300   REM - CALCULATE REAL ROOTS
310   IF S=0 THEN Q=0
320   ELSE Q=SQR(S)
330   R1= (-B-Q)/(2*A)          !ROOT 1
340   R2= (-B+Q)/(2*A)          !ROOT 2
350   PRINT "ROOTS (REAL): ";R1;", ";R2
360   RETURN

150   PRINT                     !OUTPUT BLANK LINE
160   RETURN

 60   REM - RESTART OR END PROGRAM?
 70   INPUT "MORE DATA (1-YES, 0=NO)?  "%1;N
 80   IF N<>0 THEN GOTO 20
 20   PRINT
 30   REM - ENTER COEFFICIENTS A,B,C OF A*X*X +B*X+C
 40   INPUT "COEFFICIENTS A= ";A;"  B= ";B;"  C= ";C
 50   GOSUB 100
```

```
100   REM - CALCULATE S= B*B - 4*A*C
110   S=B^2-4*A*C
120   REM - COMPLEX ROOTS?
130   IF S<0 THEN GOSUB 200    !COMPLEX ROOTS

200   REM - CALCULATE COMPLEX ROOTS
210   Q= SQR ABS (S)
220   R1 = -B/(2*A)            !REAL PART
230   R2= Q/(2*A)              !IMAGINARY PART
240   PRINT "ROOTS (COMPLEX):  ";R1;" + OR -";R2;" I"

150   PRINT                    !OUTPUT BLANK LINE
160   RETURN

 60   REM - RESTART OR END PROGRAM?
 70   INPUT "MORE DATA (1=YES, 0=NO)? "%1;N
 80   IF N<>0 THEN GOTO 20
 90   STOP
```

A RETURN statement must not be encountered unless a GOSUB statement has been executed.

"Remembering" all the return points by saving them on the GOSUB stack and never removing them can exhaust the available GOSUB stack area. The following program, which calculates N! illustrates this problem. Its use requires that N return points be remembered.

```
 10   INPUT "N= ";N
 20   GOSUB 100
 30   PRINT N,N1
 40   STOP

100   N3=N
110   N2=0
120   N1=1
130   GOTO 160
140   N3=N3-1
150   GOSUB 160
160   IF N3>1 THEN GOTO 140
170   N2=N2+1
180   N1=N1*N2
190   RETURN
```

The POP statement removes the top most previous return address from the GOSUB stack. It does not perform a return transfer to the calling routines. Execution continues at the statement following the POP statement in the internal subroutine. The POP statement is useful for exiting nested subroutines as the following example demonstrates.

```
 10 REM - MAIN PROGRAM
 20 GOSUB 100  ! CALL GET DATA
 30 .....
      .
      .
      .
      .
      .
100 REM -  ! SUBROUTINE GET DATA
110 GOSUB 200  ! CALL GET NUMBER
120 .....
      .
      .
      .
      .
      .
190 GOTO 100  ! GET NEXT DATA SEQUENCE

200 REM - SUBROUTINE GET NUMBER
210 .....
      .
      .
      .
      .
      .
250 REM - NUMBER FOUND?
260 IF NUM THEN RETURN  !   IF NUMBER - RETURN
270 REM - NO MORE NUMBERS
280 POP  ! REMOVE MOST RECENT RETURN ADDRESS
290 RETURN
```

In this example, the main program calls subroutine 100 which i
calls subroutine 200 until there is no more data. Subroutir
exits with a RETURN when data is found and a POP then RETUR
there is no more data. Program execution then continues at line

5.6.4 ON STATEMENT

Forms:

[line number] ON { <variable> / <expression> } THEN { GOTO / GOSUB } <line number>, <line number

The ON statements select the target transfer line number of a GOTO or a GOSUB from a list of statement numbers. The statement number list contains a statement number for each expected value of the expression or variable. The selection is based on the value of the expression or variable truncated to an integer. If the expression value is 1, the first line number in the list is selected. If the value is 2, the second will be executed, and so forth. The GOTO or GOSUB statement will be executed transferring control to that line. If the expresson value is less than one or greater than the number of statement numbers in the list, the transfer is not made and execution simply continues with the next statement.

Examples:

```
10  ON  J+1  THEN GOTO 15, 20, 35, 46, 70
```

When J is equal to 3, J+1 is equal to 4 and control is transferred to the fourth statement number (46). Similarly, J values of 0, 1, 2, and 4 result in jumps to statement numbers 15, 20, 35, and 70, respectively.

```
110  ON X+3 THEN GOSUB 20, 40, 80, 300
120  ON (A+5)/Z THEN GOTO 10, 30
```

When X is equal to -1, the second statement number (40) is executed next. When X is less than -2 or greater than +1, a transfer is not made and line 120 will be the next statement executed. When (A+5)/Z is equal to 2, the second statement number (30) is executed next and so forth. If the expression evaluates to a non-integer value, only the integer part is used to determine the appropriate branch point.

5.6.5 FOR/NEXT LOOPS

FOR and NEXT statements indicate the start and end of an instruction block that is to be repeatedly executed as a set. One variable takes on different values within a specified range; this variable is often used in the computation or evaluation contained in the instruction block. The FOR statement names the variable and stepping values of that variable and also specifies its initial and final values. The NEXT statement closes the program loop.

The FOR statement may have either of the following forms:

```
[line number] FOR <variable>=<expression>TO<expression>
[line number] FOR <variable>=<expression>TO<expression>STEP<expression>
```

where

variable is a simple numeric scalar variable
expression is a valid POWER BASIC numeric expression

The NEXT statement has the form:

[line number] NEXT <variable>

where

variable is a simple numeric variable

The simple variable of the NEXT statement must be the same as the FOR statement variable at the beginning of the loop.

Specification of the STEP value is optional and usually omitted. If omitted, a value of +1 is used. The step value may be any constant, variable, or expression which evaluates to a positive or negative value. Negative step intervals can be used to decrease the value of the FOR variable from one pass through the loop to the next. By using a step value of -1, the FOR variable can be made to decrease by integer values during successive loop interactions.

The following example illustrates the FOR and NEXT statements. Note that this is not a meaningful POWER BASIC program.

```
100    FOR X=0 TO 3 STEP D
200    NEXT X
300    FOR X4=(17+COS(Z))/3 TO 3*SQR(10) STEP 1/4
400    NEXT X4
500    FOR X=8 TO 3 STEP -1
600    FOR J=-3 TO 12 STEP 2
700    NEXT J
800    NEXT X
```

Note that the step size may be a variable (D), an expression (1/4), a negative number (-1), or a positive number (2). In the example with lines 300 and 400, successive values of X4 will be .25 apart in increasing order. In the next example, the successive values of X will be 8, 7, 6, 5, 4, and 3. In the last example, on successive iterations through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

If expressions are used to specify the initial, final or step-size values, they will be evaluated only once when the FOR loop is entered. Changing any of the values of the FOR loop (either the step, initial

or final values) within the FOR loop does not affect the number of times the sequence is executed except for the control variable. The control variable is assigned to the initial values when the FOR statement is entered and is incremented (if the STEP value is positive), or decremented (if the step value is negative) after each repetition of the loop sequence. The last repetition of the loop sequence is when the control variable is equal to the final value. When exiting the loop in this manner, the control variable is incremented (or decremented) one step value beyond the final value.

A pre-check is performed so that if the initial value is greater than the final value in the case of positive STEP values, the loop sequence will not be executed. Likewise, if the initial value is less than the final value and the STEP value is negative, the loop sequence will not be executed.

The control variable may be changed within the body of the loop and the latest value of the variable will be used in the exit test; however, this programming practice is not recommended.

The statement "50 FOR I=2 TO -1" without a negative step size results in the body of the loop not being executed and execution proceeds to the statement immediately following the corresponding NEXT statement. The NEXT statement must be the first item in a line for this feature to work properly.

The loop continues to be executed as long as the condition:

(step value)*(control variable)<(step value)*(end value)

remains true. If the condition:

(step value)*(start value)>(step value)*(end value)

is true when the FOR statement is first encountered, the loop will not be executed.

When the loop is being executed, the control variable is first set to the initial value and if the end criterion is not true, the loop is executed. The control variable is then incremented by the step value each time the NEXT statement is encountered and executed. The loop terminates with the control variable equal to the last value used in the loop plus the step value.

Example:

```
10  FOR I=1 TO 4 STEP 2
 .
 .
 .
```

```
80  NEXT I
90  PRINT "I=";I
RUN
I= 5
```

The NEXT statement closes the FOR loop. When it is encountered the step value is added to the control variable. If the control variable has not gone beyond the end value, control will be returned to the first statement following the FOR which opened the loop. The control variable of the loop to be closed must be specified by the NEXT statement. It is possible to place the FOR and NEXT statements on the same statement line; however; remember that statement lines are autonomous. Therefore, this type of loop structure cannot be interrupted by using the escape key since keyboard sampling is performed only between statement lines.

Also, FOR/NEXT statements on a single line or in separate statement lines will cause an error to result if, during the initial pre-check, the initial value has exceeded the final value. For example,

```
20  FOR I=10 TO 1::NEXT I
```

will result in a FOR W/O NEXT error (ERR=31).

FOR loops may be nested; i.e., one FOR loop may contain another which may contain a third, etc. If nested, however; they should not use the same control variable. When two loops are nested, one must be completely contained within the other. Overlapping is not permitted. The following structure is correct:

```
10  FOR I=1 TO 2
20   FOR J=1 TO 2
30    FOR K=1 TO 2
        .
        .
80    NEXT K
90   NEXT J
100 NEXT I
```

while the next two structures are incorrect:

```
10  FOR I=1 TO 2
20     FOR J=1 TO 2
.
.
80  NEXT I
90     NEXT J          (WRONG, loops may not overlap)

10  FOR I=1 TO 2
```

```
20     FOR I=1 TO 2
.
.
80     NEXT I
90 NEXT I              (WRONG, nested loops may not have the same
                        control variable.)
```

The following program illustrates nesting:

```
LIST
  10  REM AREA OF A TRIANGLE
  20  FOR B=6 TO 9
  30    FOR H=11 TO 13 STEP 0.5        !FIRST LEVEL OF NESTING
  40     A=B*H/2                       !SECOND LEVEL OF NESTING
  50     PRINT B,H,A                   !SECOND LEVEL OF NESTING
  60    NEXT H                         !FIRST LEVEL OF NESTING
  70  NEXT B
  80  STOP
```

This program prints the base, height, and area of triangles with bases 6, 7, 8 and 9, and heights 11, 11.5, 12, 12.5, and 13. All combinations are printed: 20 sets of data for the four bases and five height values.

All values of the variable in the inner loop are cycled through while the variable in the outer loop is set to its first value. The outer loop variable is then set to its second value and the inner loop is cycled through again. The program runs through each outer loop value this way.

Nesting of FOR/NEXT loops is permitted to a level of 10 in Configurable BASIC.

It is legal to transfer control from within a loop to a statement outside the loop, but it is never advisable to transfer control into a loop from outside. The next two examples illustrate both of these situations.

Valid transfer out of a loop:

```
20 FOR I=1 TO N
30    X=X+2*I
40    IF X>1000 THEN GOTO 100
50    NEXT I
```

Invalid transfer into a loop:

```
20  GOTO 50
30  FOR I=1 TO N
40     X=X*2*I
50     Y=Y+X/2
```

```
60  NEXT I
.       (WRONG, 50 is inside a loop)
.
```

However, it is permissable to call a subroutine from within a loop and then return from the subroutine back into the loop. The following example illustrates repetitive calling of a subroutine from inside a loop.

Example:

```
10  FOR I=1 TO N
20   X=2*I-1
30   GOSUB 150
40   Z=Z+Y
50  NEXT I
 .
 .
 .
150 IF X<>12 THEN GOTO 180
160 Y=248
170 RETURN
180 Y=200+4*X
190 RETURN
```

5.6.6 ERROR STATEMENT

The ERROR statement specifies a subroutine that will be called whenever any POWER BASIC error occurs.

Form:

[line] number ERROR <line number>

The ERROR statement enables the user to trap to an internal error processing subroutine on the occurrence of any error encountered during program execution. When an ERROR statement has been executed and an error occurs, internal control passes to the specified line number via a GOSUB statement. The statement number which contains the error is placed on top of the GOSUB stack. If the error is recoverable, a RETURN statement will resume execution at the statement following the error when it has been corrected. If the error is unrecoverable and control will not be transferred back via a RETURN, it is good programming practice to execute a POP statement to remove the line number from the top of the stack. This practice avoids unnecessary cluttering of the stack, which may cause unpredictable results. After the error trap, the system function

SYS(1) will contain the error code number and SYS(2) will contain the statement number in which the error occurred. The error codes and corresponding error messages are presented in Appendix A-1. These are necessary for processing in the error handler subroutine.

Once an error is encountered and causes transfer to the error handler subroutine, the ERROR statement flag is cleared and future errors will not be trapped unless an ERROR statement is again executed. When an ERROR statement has been executed and an error occurs, the automatic printing of the error code is suppressed.

The ERROR statement is particularly valuable when developing an application program which is to be configured into a Target POWER BASIC Interpreter and application for execution in a TM990 board system. The ERROR statement is almost required to process errors generated during program execution, since the Target POWER BASIC Interpreter consequently does not have inherent error reporting capability, and all errors are consequently fatal errors, and program execution will stop when an error is encountered. The ERROR statement may enable many of errors to be recoverable instead of fatal.

Example:

```
  10  ERROR 1000              !SET ERROR ROUTINE
  20  DIM E(2,2)
  30  INPUT A,B,C
  40  D=((A*B)+C)/B
  50  E(A,B)=0
  60  F=SQR(C)+LOG(A+C)
  70  PRINT A;B;C;D;E(A,B);F
  80  GOTO 30

1000  PRINT "ERROR =";SYS(1), "LINE= ";SYS(2)
1010  IF SYS(1)=28  THEN PRINT "DIVISION BY ZERO"::GOTO 1070
1020  IF SYS(1)=17  THEN PRINT "SUBSCRIPT OUT OF RANGE"::GOTO 1070
1030  IF SYS(1)=25  THEN PRINT "SQUARE ROOT OF NEGATIVE NUMBER::GOTO 1070
1040  IF SYS(1)=26  THEN PRINT "LOG OF NON-POSITIVE NUMBER"::GOTO 1070
1050  PRINT "FATAL ERROR"::POP
1060  STOP
1070  PRINT "INPUT VALUES AGAIN"
1080  ERROR 1000              !RESET ERROR ROUTINE FLAG
1090  POP
1100  GOTO 30
```

Statement 10 designates the subroutine, starting at statement 1000, to be the error handling subroutine. When any error occurs, control is transferred to statement 1000, and first the error number and line number are displayed on the terminal device. Next the error number is tested for various types of arithmetic errors. If the error number is

found, an appropriate error message is output, the ERROR flag is reset, and the values are asked to be input once again. If the error number cannot be found, the POWER BASIC program will stop.

5.6.7 STOP STATEMENT

The STOP statement terminates program execution at the logical end of the program. There may be one or more STOP statements in a POWER BASIC program. They may appear anywhere within the program.

Form:

```
[line number]  STOP
```

When the STOP statement is executed, the system displays the line where program execution terminated.

Example:

```
900  STOP
RUN
STOP AT 900
```

5.6.8 END STATEMENT

The END statement marks the end of a program and terminates program execution.

Form:

```
[line number] END
```

The END statement functions just as the STOP statement. It may appear as any statement within the program. When executed, the system displays the statement number where program execution terminated.

Example:

```
70  END
RUN
STOP AT 70
```

5.6.9 BYE STATEMENT

The BYE statement, when executed, terminates POWER BASIC and returns control to the TX990 operating system. During this process all files are closed, and all Logical Unit Numbers (LUNO's) are released.

Form:

[line number] BYE

Examples:

```
1000  PRINT "JOB DONE"
1005  BYE
```

5.7. INTERNAL INPUT STATEMENTS

READ, DATA and RESTOR statements are used in the following forms:

[line number] READ { <numeric variable> / <string variable> } , { <numeric variable> / <string variable> } , ...

[line number] DATA { <expression> / <string variable> / <string constant> } , { <expression> / <string variable> / <string constant> } , ...

[line number] RESTOR

[line number] RESTOR <line number>

[line number] RESTOR #<simple numeric variable>

POWER BASIC permits definition of a list of data items containing both strings and numbers within the program. Entries in this list are defined by DATA statements and accessed sequentially by READ statements. The RESTOR statement is used to move to a specific point within the list or to the beginning of the list. The RESTOR # variable statement will rewind the specified device/file.

5.7.1 DATA STATEMENT

The DATA statement contains a list of data items separated by commas. Each item in the list is either a string constant or an expression which evaluates to a numeric constant. String constants must be enclosed in quotes.

Example:

```
10 DATA 5, 3.14459,"DOE,JOHN",4*ATN(1)
```

A program may contain any number of DATA statements with no restriction on their placement within the program; however, they are

typically placed together in a data block near the beginning or end of the program. The data list will contain all of the data items from all DATA statements in the same order they are written in the program. DATA statements have no effect when encountered during execution.

5.7.2 READ STATEMENT

The READ statement assigns values from the internal data list to variables or array elements. The first READ statement executed normally starts with the first item in the data list. Reading of data items continues sequentially unless a RESTOR statement is executed. An error (READ OUT OF DATA AT XXXX) is generated when a READ statement requests the next value with the data block exhausted of data.

The READ statement specifies a list of variables or array elements whose values are to be assigned from the data list as shown below:

```
50 READ X, Y, A(5,X), $B,$C(Y)
```

The following examples illustrate use of the DATA and READ statements:

```
10 READ A,B,C,D
20 H=A*B*C*D
30 PRINT A,B,C,D,H
40 READ E,F,G
50 H=E*F*G
60 PRINT E,F,G,H
70 DATA 2,3,5,7,11,13,17
80 STOP
RUN
 2          3          5          7          210
 11         13         17         2431
```

The data in this example is supplied in one DATA statement, but is used at two different locations in the program in two READ statements. When the program encounters the first READ statement, it searches for the lowest-numbered DATA statement (which may occur before or after the READ statement). The program takes numeric values from the DATA statement in sequence associating them with READ statement variables in sequence. In the example, A is assigned the value 2, B the value 3, C the value 5, and D the value 7. The program establishes access to the next data value (11), so it may be assigned to the first variable encountered in the next READ statement. Line 20 is computed, and the newly-introduced variable H is assigned its computed value. The next READ statement at line 40 introduces three new variables. The DATA statement continues to supply data from line 70 at the

pre-established access point, so the new variables E, F, and G take on the values 11, 13, and 17. A new value for H is computed in line 50. The statement that follows prints the new values for E, F, G, and H.

The user must match numeric variables in the READ list to numeric expressions in the data list. Similarly, the user must match string variables in the READ list to string constants or string variables in the data list. An error will result if this convention is not followed.

Example:

```
10 READ A,B,$CAT
20 LET C=A+B
30 PRINT A,B,C,$CAT
40 DATA 2,3,"TEXT"
50 STOP
RUN
2          3          5          TEXT
```

5.7.3 RESTOR STATEMENT

The RESTOR statement allows more than one READ statement to access DATA statements. The RESTOR statement is used to move to a specific point in the Internal DATA list or to the beginning of the list. A "RESTOR #<variable>" statement is used to rewind the specified file/device to its beginning.

A RESTOR statement without an argument resets the pointer to the beginning of the first DATA statement. A RESTOR with an argument resets the pointer to the line number specified. The line number specified must exist but need not be the line number of a DATA statement. The next sequential DATA statement will be used.

Example:

```
70 RESTOR     (restores to the beginning of the data list)
80 RESTOR 20  (restores to the first DATA statement at or beyond
               line 20)
```

The following example program illustrates the use of RESTOR:

```
10 DATA 14,16,18
20 READ I,J,K
30 PRINT I,J,K
40 RESTOR
50 READ X,Y,Z
60 PRINT X,Y,Z
70 END
RUN
14          16          18
14          16          18
```

The RESTOR statement in this program resets the DATA pointer and transfers control to the READ statement in line 50 which then obtains data from line 10 (even though the READ statement in line 20 has used the same data). If the RESTOR statement was omitted, POWER BASIC would print an error message indicating a lack of data for the variables in the READ statement at line 50.

If the following statement is added to the example program between lines 40 and 50:

```
45 DATA 2,24,26
```

The statement at line 50 would still cause the values 14, 16, and 18 to be printed. The RESTOR statement at line 40 results in data being obtained from line 10 rather than from line 45.

If a program has no DATA or READ statements, the use of the RESTOR statement does not affect the program.

The "RESTOR #<variable>" statement will rewind the file assigned to the specified variable. The variable contains the Logical Unit Number (LUNO) assigned to a particular file. The variable is assigned the LUNO value when the file is opened by the BOPEN statement. The associated file/device will be rewound to its beginning. The file must be opened before the rewind may be performed.

When the device specified is the 733ASR cassette unit, the RESTOR statement rewinds the cassette tape to the clear area at the beginning of the tape and then moves the tape in the forward direction to the begining of tape marker, illuminating the READY indicator on the 733ASR. When the specified device is the line printer, the RESTOR statement performs a form feed operation. Performing a RESTOR on a diskette file stimulates the rewinding of the diskette file, causing the next read or write operation performed on the file to access the first record in the file. After a sequential file has been opened and record written to it, the file cannot be rewound until the file is closed via the BCLOSE or RESET statement. When the file is a relative record file, the relative record position is set to zero. The RESTOR operation is ignored by all other devices.

Example:

```
10 REM
    .
    .
    .
```

```
 40  BOPEN "CS1", I              ! OPEN CASSETTE AND ASSIGN LUNO
 50  BOPEN  "DSC:PROG/DEV",J     ! OPEN DISKETTE FILE AND ASSIGN LUNO
      .
      .
      .
110  RESTOR #I          ! REWIND CASSETTE FILE
      .                                .
      .
      .
170  RESTOR #J          ! REWIND DISKETTE FILE
```

Target POWER BASIC Interpreter: The form "RESTOR #<variable> " is not supported by the POWER BASIC Configurator, therefore the "RESTOR #<variable>" statement should not be presented in a final application which is to be configured into a customized (Target) POWER BASIC Interpreter.

5.8 TERMINAL I/O STATEMENTS

Terminal I/O Statements consist of an INPUT statement to allow keyboard input from a terminal and a PRINT statement which prints values of expressions in an output list on the output device.

5.8.1 INPUT STATEMENT

The INPUT statement is used for keyboard input from an interactive terminal into variables of the BASIC program.

Form:

[line number] INPUT <variable> {,;} <variable> {,;}

The INPUT statement performs as a READ statement with the exception that it accesses the numeric constants and strings from the external keyboard instead of from internal DATA statements. It provides all translation from character data to the internal formats of the POWER BASIC system and thus assigns input values to the variables or array elements specified in the input list. All characters are echoed as they are entered. The INPUT statement is extremely versatile and provides a means to 1) input numbers only, 2) input character strings, 3) detect control characters, 4) prompt with character strings, 5) specify maximum number of input characters, 6) specify exact number of input characters, 7) suppress carriage return/line feed, and 8) suppress prompting.

Input variables may be entered in a list separated by carriage returns. Numeric data may be represented as decimal integers, floating point, exponential, or hexadecimal values. There should be no embedded spaces within numeric values and all spaces preceding or following numeric data are ignored. For string data input, the string consists of all characters after the prompting character and up to but not including the end of the input (carriage return). The string includes all entered blanks and quotes.

The INPUT statement prompts the user with a question mark (?) for numeric only inputs and a colon (:) for character inputs. If an incorrect key is entered during input, and the terminating carriage return has not yet been entered, the RUBOUT or DEL CHAR key may be used to appropriately backspace the cursor and delete the offending character(s). The correct response may then be entered. If an illegal numeric input is entered (with a carriage return) in response to the question mark prompt, the computer will respond with a double question mark (??) and wait for correct input. The computer will continue to prompt until the user has entered all data requested.

In the following examples, a carriage return is represented by (CR), and all user responses are underlined.

Examples:

```
 40  INPUT X
 50  INPUT $A, $B
 60  INPUT $Y,Z
 70  PRINT X, $A, $B, $Y,Z
 80  STOP

RUN
 ?256 (cr)
 :CAT (cr) :DOG (cr)
 :HI (cr) ?80A (cr) ??80 (cr)
 256          CAT          DOG          HI          80

 STOP AT 80
```

In the program statement 40 outputs a question mark waiting for numeric input. The user entered the number "256" followed by a carriage return which terminated the INPUT statement of line 40. The variable X is assigned the value of "256." Next it prompts with a colon awaiting character string input. The user enters "CAT" followed by a carriage return. The computer immediately prompts with a colon awaiting the next string input. The user enters "DOG" and a carriage return which terminates this input line. The computer then prompts with a colon and the user inputs "HI" and a carriage return. Next, the computer prompts with a question mark and the user incorrectly enters "80A", an illegal numeric value. Therefore, the computer responds

with a double question mark and awaits correct input. The user enters "80" followed by a carriage return which terminates the INPUT statement. Statement 70 is then executed and outputs the values read into the variables.

An INPUT statement can be combined with a PRINT statement to prompt user response as follows:

```
20  PRINT "YOUR VALUES OF X, Y, AND Z ARE";
30  INPUT X, Y, Z
40  STOP

RUN
YOUR VALUES OF X, Y, AND Z ARE? 50 (cr) ?60 (cr) ?70 (cr)

STOP AT 40
```

Since user prompting for data input is required in most applications, the INPUT statement has been designed to permit string constants to be embedded in the INPUT statement for direct prompting output. The string constants must be enclosed by quotation marks. There may be any number of string constants within the INPUT statement separated from input variables and other string constants by commas or semicolons.

The above example may be performed as follows:

```
20  INPUT "YOUR VALUE OF X IS", X, " Y", Y, " Z", Z
30  STOP

RUN
 YOUR VALUE OF X IS? 1 (cr) Y?  2 (cr) Z? 3 (cr)

 STOP AT 30
```

Similarly for string input:

```
10  DIM N(5)
20  INPUT "WHAT IS YOUR NAME", $N(0)
30  PRINT "YOUR NAME IS ";$N(0)
40  GOTO 20

RUN
 WHAT IS YOUR NAME: JOHN  (cr)
 YOUR NAME IS JOHN
 WHAT IS YOUR NAME:
```

A semicolon may be used to perform input formatting. If a semicolon is placed at the end of an INPUT statement line, the carriage return/line feed is suppressed after processing the INPUT statement as the example below illustrates:

```
 10  INPUT "INPUT X", X;
 20  PRINT " X SQUARED="; X*X
 30  INPUT "INPUT Y", Y
 40  PRINT  "Y CUBED="; Y*Y*Y
50  STOP

RUN
 INPUT X?12 (cr) X SQUARED= 144
 INPUT Y?3 (cr)
 Y CUBED= 27

STOP AT 50
```

In line 10 the semicolon is present at the end of the INPUT sta
therefore, the carriage return/line feed is suppressed after e
the constant 12 so that "X SQUARED= 144" can be output on t
line. In line 30 a semicolon is not present so the ca
return/line feed is performed.

When the semicolon is placed before an assignment variable
INPUT list, the automatic prompting of a question mark or co
suppressed. The user may then perform his own prompting in the
BASIC Program by using PRINT statements or placing character s
in the INPUT statement.

Example:

```
 5  DIM N(3)
10  INPUT "WHAT IS YOUR EMPLOYEE NUMBER?", $N(0)
20  INPUT "WHAT IS YOUR EMPLOYEE NUMBER?"; $N(0)
30  STOP

RUN

 WHAT IS YOUR EMPLOYEE NUMBER?: 1234 (cr)
 WHAT IS YOUR EMPLOYEE NUMBER?1234  (cr)

 STOP AT 30
```

In line 10, the INPUT Statement prompted with a colon (:). In l
no prompt was issued.

5.8.1.1 SPECIFICATION ON NUMBER OF INPUT CHARACTERS

The user may limit the number of characters which can be entered
the keyboard for both numeric and string variable assignments by
the # or % operators in the INPUT statement. Use of the # ope
will specify the maximum number of characters which can be en
from the keyboard. Use of the % operator will specify the

number of characters which must be entered. The scope of both the # and % operators extend through the entire INPUT line.

Forms:

[line number] INPUT # <expression> (;) <variable> (;) ...

[line number] INPUT % <expression> (;) <variable> (;) ...

When using the # operator, the user may enter any number of characters less than the specified maximum by ending the input sequence with a carriage return. The user cannot enter more than the specified maximum number. When the maximum number of characters have been entered POWER BASIC stops accepting keyboard input, assigns the value just entered, and automatically continues to the next sequential statement or INPUT statement parameter.

Use of the % operator requires that an exact number of characters be entered. POWER BASIC waits for the exact number of specified characters to be entered and then continues to the next sequential statement or INPUT statement parameter; no carriage return (cr) is required at the end of user INPUT. If the user attempts to enter less than the specified number of characters by ending the input sequence with a carriage return, POWER BASIC will ignore the carriage return and continue to wait until the number of characters specified have been entered.

Examples:

```
10  REM THE MAXIMUM NUMBER WHICH CAN BE ENTERED IS 999
20  INPUT #3, A, B
30  STOP

RUN
 ?512 ?900

 STOP AT 30

10  PRINT "ENTER PHONE NUMBER (XXX-XXX-XXXX) ";
20  INPUT %3;A,"-";B,"-",%4;C
30  $A1=A::$B1=B::$C1=C
40  PRINT "YOUR PHONE NUMBER IS";$A1;"-";$B1;"-";$C1
50  STOP

RUN

 ENTER PHONE NUMBER (XXX-XXX-XXXX) 123-456-1234
 YOUR PHONE NUMBER IS 123-456-1234

 STOP AT 50
```

In the first example the user may enter any numbers which do not require more than three keystrokes. The range would be limited to -99 to 999. In the second example the user is requested to enter his telephone number in the format XXX-XXX-XXXX. The % symbols require the user to enter exactly the required amount of numbers. The user enters 123. The computer places the number in variable A and outputs a "-". The user enters 456, and the computer places the number in variable B and outputs a "-". The user enters 1234 to complete the sequence. Statement 30 converts the numeric inputs into strings and places them into $A1, $B1, and $C1. Statement 40 then prints the user's phone number using these string variables of the INPUT list.

5.8.1.2 INVALID INPUT CHARACTER PROCESSING

The user may detect any invalid input or control characters which are entered during both numeric and string variable assignment by using the question mark (?) operator in the input list.

Form:

[line number] INPUT ? <line number> {,;} <variable> {,;} ...

The "?" operator specifies the line number to which control is transferred via an internal GOSUB statement if a control character or invalid input is encountered during input. The scope of the "?" operator extends through the current INPUT statement only. Execution of the next INPUT statement without the "?" operator resets the invalid input function. The SYS(0) function will return the control character encountered. SYS(0) will be equal to -1 if there was an invalid input. Otherwise, SYS(0) will equal the decimal equivalent of the control character. This feature is useful for transferring control to internal subroutines by using the INPUT statement. Executing a RETURN statement in the subroutine will result in a return to the statement immediately following the INPUT statement. For example; a (control) H can be used to transfer to a routine which outputs a HELP message sequence for the user who requires additional information for the input of data.

Example:

```
 10  PRINT "INPUT VALUE ";
 20  INPUT ?100,N
 30  PRINT "NUMBER IS";N, "IT's SQUARE ROOT IS "; SQR(N)
      .
      .
      .
 90  GOTO 10
100  IF SYS(0)=-1 THEN PRINT::PRINT "NUMERIC INPUT ONLY"::RETURN
110  REM IF (CNTL) H GOTO HELP ROUTINE
```

```
120  IF SYS(0)=8 THEN GOSUB 200::RETURN
130  REM IF (CTRL) G GOTO BACKSPACE TO PREVIOUS INPUT ROUTINE
140  IF SYS(0)=7 THEN GOSUB 300::POP::GOTO 10
  .
  .
180  POP
190  RETURN

200  REM----------------HELP ROUTINE----------------
210  PRINT
220  PRINT
230  PRINT "USER INPUT ASSISTANCE"
       .
       .
       .
290  RETURN
```

Statement 20 specifies line number 100 as the entry point to the invalid input processing routine. When as invalid input occurs (in this case either a control character or non-numeric character), control is transferred to statement 100. It is first tested for invalid (non-numeric) input. If this is found to be true, it outputs the message and returns. If it is not an invalid character input, it is then tested for a Control H (08) or HELP input character. If a control H was input, the GOSUB routine may display a help menu to inform the user of all valid inputs and their functions. If not a control H character, then SYS(0) can be further tested for other special function control characters as defined by the user, such as back up to previous input, erase current input, or stop program execution.

5.8.1.3 INPUT STATEMENT CURSOR CONTROL

The user may perform screen positioning to any location of the 911VDT or 913VDT terminals before performing the specified numeric or string variable input assignment by using the @ operator in the input list. The @ operator is supported only by the Host POWER BASIC Interpreter on the FS990 computer.

Forms:

[line number] INPUT @(<exp1>, <exp2>) {,;}<variable>{,;} ....

[line number] INPUT @<$VAR> {,;} <variable>{,;}...

Cursor control via the @ operator may appear numerous times and at any position within the INPUT list. It appropriately positions the cursor

before performing the INPUT variable assignment. The INPUT and PRINT statements both support an identical set of cursor positioning commands. For a complete explanation of the cursor positioning capabilities of POWER BASIC, refer to paragraph 5.8.2.3, PRINT STATEMENT CURSOR CONTROL.

Examples:

```
10  INPUT @(0,20);I,@(5,50),J,K
20  INPUT @"C5D10R";$A
30  INPUT @"C",A1,@(10,10);B1
```

Target POWER BASIC Interpreter: INPUT Statement Cursor Control is not supported by the POWER BASIC Configurator, therefore cursor control should not be present in a final application program to be configured into a customized (Target) POWER BASIC Interpreter.

5.8.2 PRINT STATEMENT

The PRINT statement causes the values of all expressions in the list to be printed on the output terminal. Commas and semicolons are used to separate expressions and provide for print formatting.

Form:

[line number] { PRINT / ; } ⟨expression⟩ { , / ; } ⟨expression⟩ { , / ; } ... { , / ; }

The expression list may contain any numeric variable, numeric expression, string variable, string constant, or any ASCII code which is to be output to the terminal device.

String constants may be printed directly by inserting them in the PRINT statement expression list. String variables are printed by having the variable name preceded with the dollar sign designator. The following example illustrates the output of string constants and string variables.

```
100  DIM N(10)
110  $N(0)= "POWER BASIC."
120  PRINT "THE NAME OF THE LANGUAGE IS ";
130  PRINT  $N(0)
140  STOP

RUN

 THE NAME OF THE LANGUAGE IS POWER BASIC.


STOP AT 140
```

The PRINT statement may be used to directly output ASCII codes to the terminal device. The hexadecimal ASCII code must be enclosed in angle brackets, (e.g., <0A>) and may be placed anywhere with string constants or predefined string variables appearing within the PRINT statement expression list. Only the low order seven bits of the hexadecimal code will be output to the device.

Example:

```
10  PRINT "GO TO THE NEXT LINE <0A><0D> AND CONTINUE PRINTING!"
```

would generate

```
GOTO THE NEXT LINE
AND CONTINUE PRINTING
```

In a similar manner, any ASCII character can be placed into a string variable with the use of the replacement (%) operator. The % operator places the single byte value of the succeeding expression into the specified character string. The expression represents the decimal or hexadecimal ASCII representation of the byte value to be placed in the string (e.g., $A=%0AH%0DH%0H). String variable assignment using byte value insertion should always be terminated with a null (zero byte) insertion (e.g., %0AH%041H%0H). While actual character insertion into an existing string would typically not be terminated with a null insertion (e.g., $A(0;4)=%45,%41). ASCII codes may be concatented to character strings, however character strings may not be concatenated to ASCII codes in character assignments. For example, $A=$B + "YES" + %10%13 is a valid character assignment, while $A=$B + %10%13 + "NO" is an illegal character assignment. These string variables can then be output with the PRINT statement. The following example program illustrates this procedure.

Example:

```
List
 10  DIM A(10)
 20  $B=%10%13%0 !(10=LINE FEED, 13=CARRIAGE RETURN)
 30  $A(0)="GO TO THE NEXT LINE"+B+"AND CONTINUE PRINTING!"
 40  PRINT $A(0)
 50  STOP
```

would generate,

```
GO TO THE NEXT LINE
AND CONTINUE PRINTING!
```

To facilitate rapid statement entry in the edit mode, a semicolon (;) may be used in place of the word "PRINT" in any PRINT statement. Upon statement entry, the semicolon is internally translated to the "PRINT" code. Thereafter, listing of the statement will result in output of

the word "PRINT." For example"

```
10  PRINT I,J
20  ;X,Y,Z
30  ; 'THE SEMICOLON WILL LIST AS "PRINT"'
LIST

10  PRINT I,J
20  PRINT X,Y,Z
30  PRINT 'THE SEMICOLON WILL LIST AS "PRINT"'
```

In its simplest form, the expressions in the output list are separated by commas. In this form, an output line is divided into five 15-character print fields starting in columns 1, 16, 31, etc. A comma following an expression in a list is a signal to advance to the next field. Expressions separated by commas are output one expression per print field. This enables output lines to be formatted into five left justified columns within the field. Expressions may occupy more than one field, in which case the comma following the expression in the PRINT list advances the print output to the next blank field. Note that when more than five expressions are included in the output list separated by commas, the terminal device after the fifth character will automatically generate a carriage return/line feed when its buffer is full to obtain the correct five column output. Printing will continue in as many lines as are required to complete the output list. When the entire output list has been printed, a carriage return/line feed is automatically inserted after the last print item. Subsequent printing begins on the next line. For example, the following statements:

```
10  X=7
20  $NAM = "PAUL"
30  PRINT X, X+2, X+4
40  PRINT "GEORGE", "HARRY", $NAM
```

would generate the output shown below

```
 7              9              11
GEORGE         HARRY          PAUL
```

The automatic carriage return/line feed at the end of a PRINT statement may be suppressed by placing a comma at the end of the output list. Subsequent printing will begin in the next field of the same line. For example:

```
10  X=7
20  $NAM="PAUL"
30  PRINT X, X+2, X+4,
40  PRINT "GEORGE", "HARRY", $NAM
```

would generate

```
 7             9             11            GEORGE        HARRY
PAUL
```

Note that the terminals automatically generate a carriage return and line feed as occurs in the following example:

```
10  FOR I=1 TO 14
20  PRINT I;
30  NEXT I
40  STOP
```

```
RUN

 1             2             3             4             5
 6             7             8             9             10
11            12            13            14

STOP AT 40
```

More compact printing can be achieved by using semicolons rather than commas as expression separators. When followed by a semicolon numbers in the output list will print in as many characters as required to print the numbers of the expression plus one blank space added on the left. However, strings in the output list will print in exactly the end of an output list, the last item will print in a short field as just described and subsequent printing will begin immediately after that field. For example:

```
10  S1=95
20  S2=87
30  S3=92
40  PRINT "SCORES AND NAME:";S1;S2;
50  PRINT S3; " JOE DOE"
```

would generate

```
SCORES AND NAME: 95 87 92 JOE DOE
```

Another example:

```
10  FOR I=1 TO 14
20  PRINT I;
30  NEXT I
40  STOP
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14

STOP AT 40
```

Note that both semicolons and commas may be used to separate expressions in any PRINT statement and that the print position of the next expression will depend on the separator (semicolon or comma) used to delimit the expressions. The following example illustrates the use of both delimiters in a single PRINT statement.

```
10  H=98
20  L=60
30  A=79
40  PRINT "HIGH= ";H,"LOW= ";L,"AVERAGE= ";A
```

would generate

```
HIGH=  98        LOW=  60        AVERAGE=  79
```

A PRINT statement without an expression list is a valid statement. Execution of this statement results in the output of one blank line as the example below illustrates.

```
10  PRINT "THERE SHOULD BE TWO BLANK LINES BETWEEN HERE AND"
20  PRINT
30  PRINT
40  PRINT "HERE!"
```

would generate

```
THERE SHOULD BE TWO BLANK LINES BETWEEN HERE AND


HERE!
```

5.8.2.1 PRINT FORMATTING

The PRINT statement may be used to specify the exact print format for the output of numeric expressions. The pound sign (#) within a PRINT statement followed by a hexadecimal formatting character or a decimal formatting string provides this capability.

Forms:

[line number] PRINT ⟨#⟩ ⟨exp⟩ {;} ....

[line number] PRINT ⟨#,⟩ ⟨exp⟩ {;} ....

[line number] PRINT ⟨#;⟩ ⟨exp⟩ {;} ....

[line number] PRINT ⟨#⟩⟨string constant⟩ ⟨expression⟩ {;}

[line number] PRINT ⟨#⟩⟨string variable⟩ ⟨expression⟩ {;}

The formatting function may appear anywhere within the parameter list of the PRINT statement. The parameters within the PRINT statement are separated by commas or semicolons as explained in the PRINT statement (paragraph 5.8.2). A separator appearing at the end of the parameter list will force subsequent printing to continue on the same line just as in the PRINT statement.

A format designator (#) followed by a semicolon, comma, or space is used to output hexadecimal values in either byte, word, or free format, respectively. These format specifiers convert to hexadecimal the numeric constant, variable or expression immediately following the specifier. The scope of the hexadecimal format specifier is for the first constant, variable, or expression only and not for the entire line as in the case of print formatting using a string image. Subsequent values will be printed in free format decimal representation.

The "#;" specifier converts the value and outputs the hexadecimal result as a single byte with no preceding or trailing blanks or zeros and without the "H" character. Only the least significant byte will be output for values which require more than one byte for their hexadecimal representation.

The "#," specifier converts the value and outputs the hexadecimal result as a full word (two bytes) with no preceding or trailing blanks or zeros and without the "H" character. The least significant two bytes will be output for values requiring more than one word for their hexadecimal representation.

The "#" specifier by itself converts the value and outputs the result in hexadecimal free format representation. The hexadecimal result occupies as many digit positions as required to print the number. It is preceded with a zero (0) and followed by the "H" character.

The following examples illustrate hexadecimal output formatting. The user will terminate the entry line with a carriage return. POWER BASIC outputs are designated by underlining.

```
PRINT #;1;"  ";#,1;"  "; #1   01  0001  01H
PRINT #;31;"  ";#,31;" "; #31;" ";31 1F  001F  01FH  31
LET A=106
PRINT #;A;"  ";#,A;"  ";#A;"  ";A   6A  006A  06AH  106
```

Numeric decimal formatting is designated within a PRINT statement parameter list by a print format specifier (#) followed by a format constant or string variable. The format string may be either a string constant enclosed in quotes which directly contains the formatting string, or a string variable which has previously been assigned the formatting string.

The format string indicates the final printed image of how the numeric expressions specified within the PRINT statement parameter list are to be output. Fields are reserved for printing numeric data by forming output images of the printed results. Special characters are used within the format string to indicate these results.

Several formatting strings may be interspersed within a single PRINT statement parameter list. Numeric output values use the last defined print format in that statement line for their output. Exit from a PRINT statement line resets the formatting flag with subsequent numeric values printed in free format. That is, the range of print formatting is limited to the print statement line in which it is located. Subsequent PRINT statements each require their own print format specifier (#) and string.

Text to be output may be interspersed within the formatting string so long as it contains none of the special characters used for print formatting.
The special characters used in the formatting string are shown in Table 5-2.

When using print formatting, floating point numeric values are rounded to the number of decimal places specified by the format string. A formatting error occurs if a numeric value is inconsistent with the specified formatting string or if the integer portion of a value requires more digits than specified by the format string. This is indicated to the user by filling the entire output field with asterisks (*).

The following paragraphs and examples explain the use of formatting characters. In these examples single quotes(') are embedded within the format field so the actual printed results can be shown more clearly.

TABLE 5 - 2

FORMATTING STRING CHARACTERS

CHARACTER	FUNCTION	EXAMPLE
.	Decimal point specifier	PRINT #"999.99"25.32; Ø25.32
^	Translates to decimal point	PRINT #"999 00"1000; Ø10.00
,	Suppressed if before significant digit	PRINT #"99,999.99"100;ØØØ100.00
9	Digit holder	PRINT #"9999"123; Ø123
0	Digit holder or forces zero	PRINT #"9999.999".234;ØØØ0.234
$	Digit holder & floats $	PRINT #"$$$.99"8; Ø$8.00
S	Digit holder & floats sign	PRINT #"SSS.99" -6; Ø-6.00
E	Sign holder after decimal	PRINT #"990.99E"-150.75; 150.75
<	Digit holder before decimal & floats on negative number	PRINT #"<<< .00>" 500;500.00
>	Appears after decimal if negative	PRINT #"<<<.00>" -50;<50.00>

In practice these quotes typically would not be used. The user may xecute these examples from the keyboard by entering the example through the final semicolon (;), inclusive, and then terminating the entry line with a carriage return. All POWER BASIC outputs are underlined.

POWER BASIC will respond with the formatted results output between the quotes.

The "9" and "0" formatting characters are used as digit holders. The period (.) character specifies the decimal point position on output.

```
PRINT #"'99'"  5;'5'
PRINT #"'999.00'"25.32;' 25.32'
PRINT #"'99.0'" 15.575;'15.6'
PRINT #"'99.0'" 101.25;******
```

The " " formatting character also forces a zero if a non-significant digit is output at that position.

```
PRINT #"'999.00'"28;' 28.00'
PRINT #"'990.00'".153;'  0.15'
PRINT #"'990.000'".75;'  0.750'
PRINT #"'990.000'" 1047.23;**********
PRINT #"'000-00-000'" 3021; '000-03-021'
```

The "^" formatting character translates to a decimal point upon output wherever it is located in the format field. For example, this is useful when performing monetary calculations in pennies and then translating the results to dollars and cents on output.

```
PRINT #"'999^00'"200;'  2.00'
PRINT #"'999^00'"2532;' 25.32'
PRINT #"'999^00'"12000;'120.00'
```

The comma (,) formatting character inserts a comma in the output numeric value; however, it is suppressed if there are no significant digits to the left of its position in the output value. Typically, it is used to separate groups of three decimal digits, (e.g., 1,000 and 1,000,000).

```
PRINT #"'99,990.00'"3529.87; ' 3,529.87'
PRINT #"'99,990.00'" 903; '   903.00'
PRINT #"'99,990.00'"10.2333;'    10.23'
PRINT #"'99,990.00'"100256.72;**********
```

The dollar ($) sign formatting character is used to output the dollar sign with the numeric output value. It is a digit holder and also "floats" to the position immediately to the left of the most significant digit of the output value.

```
PRINT #"'$$$.00'"25.32;'$25.32'
PRINT #"'$$$.00'".50;'  $.50'
PRINT #"'$$$.00'"100;'100.00'
PRINT #"'$$$.00'"1000;********
PRINT #"'$,$$$.00'"1.52; '   $1.52'
PRINT #"'$$,$$$.00'" 9536; '$9,536.00'
```

The "S" formatting character is used to output a signed numeric value. A minus sign (-) is output for a negative number and blank for a positive number. The "S" character is a digit holder and "floats" the sign of the numeric value to the position immediately to the left of the most significant digit of the output value.

```
PRINT #"'SSS0.00'" 208.79; ' 208.79'
PRINT #"'SSS0.00'" -20.79; ' -20.79'
```

If the user attempts to output a negative number without using the "S" formatting character, the number will be output as a positive number.

The "E" formatting character is used to output a signed numeric value with the sign appearing to the right of the decimal point. It functions only as a sign holder and is not a digit holder.

```
PRINT #"'990.00E'" 32.253; ' 32.25 '
PRINT #"'990.00E'"-32.253; ' 32.25-'
PRINT #"'990.00E'" -.50; '  0.50-'
```

The "<" and ">" formatting characters are used in another form of outputting negative numbers. They typically are used together in the formatting string. The "<" character is a digit holder and appears before the decimal point. The ">" character appears after the decimal point and is only a sign holder. On the output of a negative number both the "<" and ">" characters are output with the string. The "<" character will float on a negative number to the position immediately to the left of the most signficant digit of the output value. The ">" character will appear at its position to the right of the decimal point on a negative number. When outputting a positive number, neither the "<" nor ">" character will be output in the string.

```
PRINT #"'<<<<<<.00>'" 1250; '  1,250.00 '
PRINT #"'<<<<<<.00>'" -1250; ' <1,250.00>'
PRINT #"'<<.00>'" .20; '   .20 '
PRINT #"'<<.00>'"-0.2; '  <.20>'
```

The following sample program further illustrates the results of print formatting. When this program is executed the user is requested to enter a numeric value and formatting string. POWER BASIC then outputs the number using the user supplied print formatting string.

```
100  DIM F(5)
110  INPUT "INPUT NUMBER"N"  FORMAT"$F(0)
120  PRINT "'"#$F(0);N"'"
130  GOTO 110

RUN
INPUT NUMBER? 1  FORMAT:  999,990.99
'      1.00'
INPUT NUMBER? 123456   FORMAT:  999,990.99
'123,456.00'
INPUT NUMBER? 529728761  FORMAT:  000-00-0000
'529-72-8761'
INPUT NUMBER? 2335.34  FORMAT:  $$$,$$$,$$$.99E
'    $2,335.34 '
INPUT NUMBER? -234.56  FORMAT:  SSSSS.99
' -234.56'
INPUT NUMBER? -2335.34  FORMAT:  $$$,$$$,$$$.99E
'    $2,335.34-'
INPUT NUMBER? 1234556  FORMAT:  999,999
'*******'
INPUT NUMBER? 123  FORMAT: <<<<<.<0.99>
'    123.00'
INPUT NUMBER? -1234  FORMAT: <<<<<.<0.99>
'<1,234.00>'
```

5.8.2.2 TAB

Output formatting can also be controlled by use of the TAB function.

Form:

TAB (<expression>)

The expression in the TAB function specifies the horizontal column position where the print item following the TAB will begin printing. The TAB function may contain any expression as its argument. The expression is evaluated and its integer portion used. If the result is greater than the line size, the specified print item will be printed on the next output line. If the column specified by the integer part of the expression has already been passed in the current print line, the TAB function will be ignored and the print item will be output at the current position in the print line. The TAB function may be used to format output into columns on the output device.

Examples:

```
10  PRINT "BIG"; TAB(20);"SPACE"
```

will generate

```
BIG                  SPACE
```

while:

```
10  PRINT TAB(20); "SPACE";TAB(1);"BIG"
```

will generate

```
SPACEBIG
```

In the first example, the string "BIG" is output starting in column 1. The TAB function advances the printer to column 20 and outputs the string "SPACE". In the second example, the TAB Function advances the printer to column 20 and outputs the string "SPACE". The TAB (1) attempts to return the printer to column 1 in the print line. Since that column position has already been passed, the string "BIG" is output immediately following "SPACE" (the current position on the print line).

5.8.2.3 PRINT STATEMENT CURSOR CONTROL

In conjunction with the PRINT statement, the user may position the cursor to any location on the video display terminal log device.

Cursor control is performed on the Host POWER BASIC Interpreter with the use of the @ operator. Note that cursor positioning is supported only by the Host POWER BASIC Interpreter and not by the Target POWER BASIC Interpreter. Therefore cursor positioning should not be included in a final application program which is to be configured into a customized (Target) POWER BASIC Interpreter for execution in a TM990 board system.

Forms:

[line number] PRINT @(<exp1>,<exp2>) {,;} <variable> {,;} .... {,;}

[line number] PRINT @<$var> ; <variable> {,;} .... {,;}

Cursor control via the @ operator may appear numerous times and at any position within the PRINT statement. It appropriately positions the cursor before performing the output of any succeeding variables, expressions, or string constants.

By using the "@(<exp1>,<exp2>)" form, the cursor can be positioned to any location on the 911 or 913 VDT terminals as specified by exp1 and exp2. <Exp1> specifies the ROW position, while <exp2> specifies the COLUMN position where the cursor is to be positioned. The 911 video display terminal has 24 rows of 80 characters (ie., 24 ROWs and 8 columns), while the 913 video display terminal has only 12 rows of 80 characters (ie., 12 ROWs and 80 COLUMNs). The COLUMN values expected by POWER BASIC range from 0 to 79, while the ROW values range from 0 to 11 and 0 to 23. The expressions are evaluated and their integer portions are used. The user should limit the range of <exp1> and <exp2> to the values given above.

Examples:

```
10  DIM STR(10)
20  INPUT @"C";"INPUT Y-POSITION ";Y
30  INPUT "INPUT X-POSITION ";X
40  INPUT "INPUT STRING ";$STR(0)
50  PRINT @"C";@(Y,X);$STR(0)
60  INPUT @(0,0);"TYPE C/R WHEN READY TO CONTINUE ";A
70  GOTO 20
80  STOP
```

This example will request user input of the X and Y positions, and the character string to be output. The user enters these values, and then the Host POWER BASIC Interpreter will appropriately position the

cursor and output the character string. It then returns the cursor to the upper left hand corner and waits for the user to enter a carriage return to continue for the next position parameters and character string.

```
 10  $STR="*"
 20  INPUT @"C";"HOW MANY ROWS ON YOUR TERMINAL, 12 or 24? ";ROW;@"C"
 30  ROW=ROW-1
 40  FOR I=1 TO 200
 50   PRINT @(ROW*RND,(80-LEN($STR))*RND);$STR
 60  NEXT I
 70  T1=TIC(0)
 80  IF TIC(T1)<5 THEN GOTO 80
 90  PRINT @"C"
100  GOTO 40
110  STOP
```

This program outputs an asterisk (*) to two hundred random locations on the screen of the 911 or 913 VDT's, waits for 5 seconds, and then clears the screen and repeats the sequence.

By using the "@<$var>" form, the user can specify several additional screen cursor positioning commands (eg.,HOME, CLEAR, and BEGINNING). They also permit VDT independent cursor positioning by moving the cursor to the RIGHT, LEFT, or DOWN. The <$var> may be either a string constant or string variable specifying the sequence of cursor control commands. The special character codes used in the <$var> are as follows:

CODE	ACTION
B	Move cursor to beginning of line
C	Clear screen and move cursor to HOME position
D	Move cursor down
H	Move cursor to HOME position
L	Move cursor to left
R	Move cursor to right

Any of these codes may be preceeded by positive integer representing the number of times the following code is to be repeated when the $var is output.

Examples:

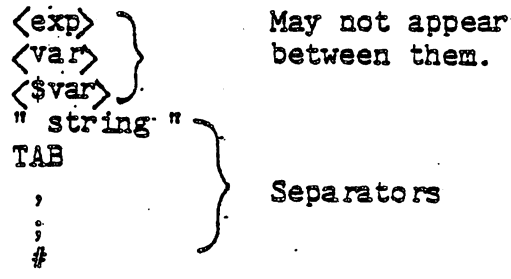
```
 10  DIM SCR(5)
 20  $SCR(0)="C5D10R"
 30  PRINT @$SCR(0);"5 DOWN AND 10 TO THE RIGHT"
 40  PRINT @"H5D20R10L";"5 DOWN AND 10 TO THE RIGHT"
 50  PRINT @"C";@(5,10);"5 DOWN AND 10 TO THE RIGHT"
 60  STOP
```

The three PRINT statements of this example will produce the same results, in that the cursor will be positioned at the 5th row, and 10 column before the string is output. Statements 30 and 50 will however, clear the screen before the cursor is positioned.

Target POWER BASIC Interpreter: PRINT statement cursor control is not supported by the POWER BASIC Configurator. Therefore, cursor control should not be presented in the final application to be configured into a customized (Target) POWER BASIC Interpreter.

5.8.2.4 SUMMARY - PRINT STATEMENT RULES

The PRINT statement may contain the following elements; any number of times and in any sequence within the expression list. The only restriction is that no two expressions (exp) may appear together without a separator between them. Valid separators are a comma (,), a semicolon(;), or a pound sign (#). An expression is defined as any arithmetic combination of numeric constants, variables, or functions. For example, PRINT 3+5 2*SQR(A), is an illegal statement.



Most users insert redundant semicolons (;) and parenthesis within the expression list of PRINT statements to facilitate readability and clarity. However, the experienced user may eliminate many of these redundant characters to save memory area and increase the speed of interpreter execution.

The following examples show the typical format of a PRINT statment:

```
100   PRINT "A=";A;"B=";B
110   PRINT A;TAB(10);"HI";#"999.99";A;TAB(25);B
120   PRINT 25;$B;"STRING";B
130   PRINT $A;$B
140   PRINT B;$A;C
```

These statements could be altered to:

```
100   PRINT "A="A"B="B
110   PRINT A TAB 10 "HI"#"999.99"A TAB 25;B
120   PRINT 25 $B "STRING" B
```

```
130   PRINT $A $B
140   PRINT B $A;C
```

The following examples illustrate invalid PRINT statement expression lists:

```
100   PRINT A B 25
110   PRINT 250 SQR(A)
120   PRINT $A B 15
130   PRINT 5*SQR(A) A*B/C
```

These statements must be written as:

```
100   PRINT A;B;25
110   PRINT 250; SQR(A)
120   PRINT $A;B;15
130   PRINT 5*SQR(A);A*B/C
```

These techniques should only be used in programs which will never be read by other than expert POWER BASIC programmers. Saving space and time at the expense of program clarity may cost more in the long run than you are willing to pay.

5.8.3 DIGITS STATEMENT

The DIGITS statement specifies the number of significant digits to be printed in format free output.

Forms:

[line number] DIGITS <expression>

The expression may be any numeric constant, variable, or expression which is evaluated and its integer portion used. The range is from 1 to 11 for the expression of the DIGITS statement. If the expression is outside these limits, a DIGITS OUT OF RANGE error will result. From the DIGITS statement on in the program, all format free printing will display a maximum of the specified number of decimal digits until another DIGITS statement or SAVE statement is executed. When saving a program, POWER BASIC first internally executes a DIGIT 8 statement and then proceeds with saving the program. Therefore, in programs which are to be SAVEd, all constants declared in the program should have no more than 8 significant decimal digits.

Examples:

```
      DIGITS 3                      (In keyboard mode)
      PRINT  4*ATN(1); 3.14         (In keyboard mode)
10    DIGITS 8
```

```
20  PRINT  4*ATN(1)
30  DIGITS 11
40  PRINT  4*ATN(1)
50  STOP

RUN

3.1415926
3.1415926595

STOP AT 50
```

When using format free output the following rules for printing numbers will assist the user in interpreting the printed results.

1. If the result is an integer number with an absolute value less than 10^(DIGITS-1), POWER BASIC will output the number in decimal format preceded by a minus sign if negative or a blank if positive. (i.e., if DIGITS = 7 then $x\ 10^6$.)

2. If the result is a floating point number with an absolute value less than 10^(DIGITS-1) but greater than 10 -(DIGITS-1), POWER BASIC will output the number in decimal format prededed by a minus sign if negative or a blank if positive (i.e., if DIGITS = 7 then $10^{-6} > |x| > 10^6$). If the number of significant digits is more than what is specified by the digits statement, the number will be rounded to the specified number of digits before output. Zeros trailing the decimal point are suppressed.

3. If the number is an integer or a floating point number with an absolute value greater than 10^(DIGITS-1) or less than 10^-(DIGITS-1), the number is rounded to the number of digits set by the DIGITS statement (if required), and is printed in exponential format. POWER BASIC prints exponential results as:

(a) a blank

(b) a minus sign if negative

(c) the first digit of the number

(d) the decimal point

(e) the remaining digits

(f) the letter "E" (indicating exponent)

(g) a minus sign if the exponent is negative

(h) the exponent value printed as two digits

Examples: (Using DIGITS 7 for output)

```
PRINT 123; 123
PRINT -54.546; -54.546
PRINT .0000000123; 1.23E-8
PRINT 32437580259; 3.243758E10
```

Target POWER BASIC Interpreter: The DIGITS statement is not supported by the POWER BASIC Configurator, therefore the DIGITS statement should not be present in any final application program to be configured into a customized (Target) POWER BASIC Interpreger.

5.8.4 OUTPUT CONTROL STATEMENTS

The UNIT and SPOOL statements are used to direct output to the various devices on the system.

5.8.4.1 SPOOL STATEMENT

The SPOOL statement is used to indicate the secondary output device controlled by the UNIT statement.

FORM:

[line number] SPOOL <exp> TO <numeric variable>

The SPOOL statement in combination with the UNIT statement directs output to one or more specified devices on the system. The expresison is evaluated and the result is the unit number to which spooling is assigned. The numeric variable designates the secondary device to which output will be sent. The numeric variable contains the logical unit number (LUNO) returned by a BOPEN statement, and represents the TX990 pathname of the output device/file. Spooling is typically performed to the line printer, a diskette file, and the 733 ASR Printer.

The current version of the Host POWER BASIC Interpreter supports output spooling to only one secondary device at a time. This implies that the valid unit numbers of the UNIT statement are 0,1,2,3. Unit 0 stops all terminal output, Unit 1 is assigned to the 911 VDT or 913 VDT terminal device, Unit 2 is the spooling unit number, and Unit 3 directs output to both Units 1 and 2. Therefore, the expression of the SPOOL statement must always evaluate to 2 to be a valid spooling unit number. For examples of the SPOOL statement see the UNIT statement below, Paragraph 5.8.4.2.

Target POWER BASIC Interpreter: The SPOOL statement is not supported by the POWER BASIC configurator, therefore the SPOOL statement should not be present in any final application program which is to be configured into a customized (Target) POWER BASIC Interpreter.

5.8.4.2 UNIT STATEMENT

The UNIT statement designates the device or devices to which all subsequent output will be sent. All output will be directed to the device(s) selected, including LIST output, BASIC command/statement output, and all program generated output. The devices/files will remain selected for output until a subsequent UNIT statement is encountered.

Forms:

[line number] UNIT <expression>

The UNIT statement is one of the POWER BASIC statements which produces similar results, but functions differently on the Host POWER BASIC Interpreter than on the Target POWER BASIC Interpreter. These differences will be presented below in their respective sections.

Host POWER BASIC Interpreter:

The expression may be any numeric constant, variable, or expression which is evaluated and its integer portion is used. The valid range of UNIT numbers area 0,1,2, and 3. A UNIT value of 0 will stop all output to any devices, a UNIT value of 1 will direct all output to the terminal device, an UNIT value of 2 directs all output to the device/file designated by the SPOOL statement, and a UNIT value of 3 directs all output to both the terminal device and the device/file designated by the SPOOL statement.

Examples:

```
10  BOPEN "LP",LP
20  SPOOL 2 TO LP
30  UNIT 3
40  REM  ALL SUBSEQUENT OUTPUT WILL BE DIRECTED TO BOTH THE
50  REM  TERMINAL DEVICE AND THE LINE PRINTER.
```

This example would direct any subsequent output to both the terminal device and the line printer, until another UNIT statemnet is encountered.

```
BOPEN "DSC2:PROB/OUT",OUT
SPOOL 2 TO OUT
UNIT 2
RUN
```

This example would direct all output of the POWER BASIC program executed by the RUN command to the diskette file named:PROG/OUT. All program inputs would still be from the log device, but no printing would occur on the log device until execution of a UNIT 1 or UNIT 3

statement.

Target POWER BASIC Interpreter:

The expression may be any numeric constant, variable, or expression which is evaluated and its integer portion is used. The valid range of UNIT numbers are 0,1,2,and 3. The UNIT statement is used in a Target POWER BASIC application which requires output to both ports of a TM990/101M microcomputer board. The UNIT statement can direct output to either or both of the serial interfaces present on the TM990/101M. (Note that the TM990/100M microcomputer board has only one serial I/O port, and therefore the UNIT statement will only affect the output to port A of this board.)

The UNIT number assignments are as follows:

UNIT	SERIAL I/O PORT
0	NEITHER PORT A NOR B
1	ONLY PORT A (CRU = 0080_{16})
2	ONLY PORT B (CRU = 0180_{16})
3	BOTH PORTS A AND B

Note that the baud rate of Port B must be set by executing the "BAUD 1,X" statement as described in paragraph 5.8.5 below before Port B should be referenced in a UNIT statement (eg., UNIT 2 or UNIT 3).

Examples:

```
10  BAUD 1,4
20  UNIT 3
30  PRINT 4*ATN(1)
```

In this example all subsequent output will be directed to both ports A and B of the TM990/101M microcomputer board.

The UNIT statement may be used to momentarily stop output to the primary terminal device. This would be useful to disable echoing of characters on an INPUT statement as follows:

```
50  UNIT 0           !DISABLE BOTH PORTS A AND B
60  INPUT ;$CHR(0)   !INPUT VALUES FROM TERMINAL WITHOUT ECHO
70  UNIT 1           !ENABLE PORT A
```

5.8.5 BAUD STATEMENT

The BAUD statement is used to set the baud rate of the serial I/O port(s) of the Target TM990 system via program statement control.

Forms:

[line number] BAUD <expression 1>, <expression 2>

The BAUD statement sets the baud rate of the serial I/O port(s) of only the Target TM990 system. It does not affect POWER BASIC program execution on the Host POWER BASIC Interpreter. The BAUD statement, when encountered on the Host POWER BASIC Interpreter, will be checked only for syntax, and will otherwise be skipped, and the next statement will be executed. The BAUD statement is used in application programs to be configured into a customized BASIC Interpreter to appropriately initialize the BAUD rates of the serial I/O port(s) of the TM990 boards when powered up and RESET.

The BAUD statement, when encountered on the Target POWER BASIC Interpreter will perform as follows.

The BAUD statement will initialize the TMS9902 Asynchronous Communications Controller of either port A or B as specified by expression 1, to the baud rate specified by expression 2.

Expression 1 will be evaluated and its integer portion will be used. A zero value for expression 1 will select port A (CRU address of >80) of the TM990/100M or TM990/101M microcomputer board, while a non zero value will select port B (CRU address of >180) of the TM990/101M microcomputer board.

Expression 2 will be evaluated and its integer portion will be used. The table below presents the valid range for expression 2 and the corresponding baud rates.

Expression Value	Baud Rate
0	19,200
1	9600
2	4800
3	2400
4	1200
5	300
6	110

Examples:

```
10  BAUD  0,0
20  BAUD  1,2
```

Host POWER BASIC Interpreter: The BAUD statement is only checked for syntax when executed on the Host POWER BASIC Interpreter, and is otherwise skipped and the next statement is executed.

5.9 INTERRUPT PROCESSING

Three statements are supplied for interrupt processing using a BASIC language subroutine. These statements have the following form:

```
[line number]  IMASK  <expression>
[line number]  TRAP   <expression>  TO  <line number>
[line number]  IRTN
```

The IMASK statement allows the user to control the interrupt mask of the processor. The TRAP statement associates an interrupt level with the statement number entry point for the interrupt processor subroutine written in BASIC. The user will return from this subroutine with the IRTN statement.

The interrupt processing statements are used only in the Target POWER BASIC Interpreter. They are non-operations (NOPs) in the Host POWER BASIC Interpreter since the TX990 Operating System and the FS990 system use many of the interrupt vectors for its interrupt driven device interfaces. The interrupt processing statements IMASK and TRAP are checked for syntax when encountered in a program, however they are otherwise skipped, and the next statement is executed. The IRTN statement, on the otherhand, performs as a RETURN statement, that is, it removes the top item off the GOSUB stack, and returns to the point in the BASIC program from where it was called. This is useful in testing BASIC Interrupt processing routines by selectively executing a GOSUB to the entry point of the interrupt handling routine, and then returning using the IRTN statement as would be done in the final application.

The following sections explain how the interrupting processing statemetns function in the Target POWER BASIC Interrupt and application.

5.9.1 IMASK STATEMENT

The IMASK statement is used to control the interrupt mask of the TMS9900 microprocessor. The TMS 9900 microprocessor employs 16 interrupt levels with the highest priority level being 0, and the lowest 15. Level 0 is reserved for the RESET function; all other levels may be used for external devices. The external levels may also be shared by several device interrupts, depending on system requirements. Since the reset sequence at power-up sets the interrupt mask to zero, the appropriate interrupt mask must be set before any interrupts will be acknowledged.

Note that if the current level is less than 3, setting of the system time by using the TIME statement will result in the interrupt mask being set to level 3. Likewise, if the real time clock is being used (located at interrupt level 3), and if the mask is subsequently set to

less than 3, the clock interrupts will no longer be acknowledged and real time will be destroyed.

All interrupts before they reach the TMS9900 CPU are first masked by the TMS9901 Programmable Systems Interface. To prevent unwanted interrupts from being acknowledged, the user must appropriately set the interrupt mask of the TMS9901 to select all interrupt levels which are to be processed. This is performed via the CRU interface using the BASE, CRB, and CRF POWER BASIC statements.

Examples:

```
10   IMASK  15  ! SET MASK TO 15
20   IMASK  0EH ! SET MASK TO 14
30   A=0AH      ! SET A TO 10
40   IMASKA     ! SET MASK TO VALUE OF A
```

Host POWER BASIC Interpreter: The IMASK statement is only checked for syntax when executed on the Host POWER BASIC Interpreter, and is otherwise skipped and the next statement is executed.

5.9.2 TRAP STATEMENT

The TRAP statement is used to define the entry point of the interrupt subroutine for a given interrupt level. The level "expression" may be any valid POWER BASIC expression whose integer portion is used and whose value is masked to the least significant 4 bits. The "line number" specifies the entry point for the interrupt servicing routine.

The TMS9900 Microprocessor continuously compares the incoming interrupt code with the interrupt mask as set by the IMASK statement. When the level of the pending interrupt is less than or equal to the current mask level (a higher or equal priority interrupt), the processor recognizes the interrupt. Note that interrupts are acknowledged immediately and the mask value is appropriately set, but that the BASIC interrupt processor will only be entered following completion of the currently executing statement. A statement is terminated by either an end of line or by the double colon (::) statement separator.

After an interrupt occurs, the interrupt mask is set such that the current interrupt level is disabled and only higher priority levels are enabled (the mask value set to one less than the level of the interrupt being serviced). Should a higher priority interrupt occur while servicing an interrupt, the interrupt processor will complete the current statement and then transfer to service the higher priority level interrupt. Upon completion of higher level processing, an IRTN statement is used to terminate the higher level interrupt servicing routine, restore the previous service routine parameters to the processor, and return control to the previous service routine at the

point which the higher priority interrupt occurred to complete processing of the lower-priority interrupt. Should a lower level interrupt occur, it will remain pending until the interrupt mask is raised to allow the interrupt.

Note that interrupt levels 0 (RESET) and 3 (clock) are reserved and should not be serviced by the TRAP statement.

Examples:

```
10   TRAP   5   TO 500  !  ASSIGN LEVEL 5 TO LINE 500
20   TRAP  0EH  TO 100  !  ASSIGN LEVEL 14 TO LINE 100
30   A=200              !  SET LINE
40   B=0CH                 SET LEVEL
50   TRAP B TO A        !  ASSIGN LEVEL 12 TO LINE 200
```

Host POWER BASIC Interpreter: The TRAP statement is only checked for syntax when executed on the Host POWER BASIC Interpreter, and is otherwise skipped and the next statement is executed.

5.9.3 IRTN STATEMENT

The IRTN statement is used to return from an interrupt servicing processor. IRTN is the last statement and terminates the interrupt servicing processor. It will restore the program environment existing when the interrupt was taken, and will return control to the previous routine at the point at which the interrupt occurred.

Examples:

```
190  IRTN                 ! RETURN FROM INTERRUPT LEVEL PROCESSING
```

Host POWER BASIC Interrupt: The IRTN statement when executed on the Host POWER BASIC Interpreter performs like the RETURN statement, and removes the top item off of the GOSUB stack and returns to the position in the BASIC program from where it was called.

5.9.4 ASSEMBLY LANGUAGE PROCESSORS

There are times when it maybe necessary or advisable for the interrupt processor to be written in assembly language. This may be accomplished in two ways when using the Target POWER BASIC Interpreter. The first is to use the TRAP statement and the CALL statement to access the assembly language routine. The second is to modify the interrupt transfer vectors for the desired interrupt level so that an interrupt will transfer to the assembly language routine directly.

Low-order memory, addressed as 0 through 3F, is reserved for the transfer vectors used by the interrupts. When an interrupt request at

an enabled level occurs, the contents of the transfer vector corresponding to the level are used to enter the subroutine to serve the interrupt.

The reserved memory locations are shown in the Interrupt Level Data table (Table 5.3). Two memory words are reserved for each interrupt level. The first of the two words for a given level contains an address that is placed in the WP when the interrupt is requested and enabled. The second contains the entry point of the interrupt subroutine for that level; its contents are placed in the PC.

To install an assembly language interrupt processor into the Target POWER BASIC Interpreter, the user must modify the object module produced by the TXDS Link editor before programming this object module into EPROM. The transfer vector for the desired interrupt level must be modified to reflect the new workspace pointer and the new entry point for the interupt level must be modified to reflect the new workspace pointer and the new entry point for the interrupt handling routine. The object module produced by the TXDS Link Editor has its origin at hex address 0000_{16}, therefore the transfer vector locations are appropriately offset from the beginning of the object module (ie., offsets 0_{16} through $3F_{16}$ are the transfer vectors). The EPROM set may then be programmed with this modified object module containing the desired interrupt transfer vector.

All assembly language interrupt processors must supply their own workspaces, therefore RAM must be allocated for this purpose. During power up reset, POWER BASIC will automatically size all available contiguous RAM from hex $FFFE_{16}$ on down for its own use. Consequently, the user must supply a non-contiguous RAM area for the workspaces in the Target POWER BASIC system.

Note that interrupts serviced by assembly language processors are handled transparent to POWER BASIC; that is, a) the transfer to the interrupt service routine is external to the POWER BASIC processor (POWER BASIC has no knowledge an external interrupt has occurred), and b) the transfer is made immediately upon receiving the interrupt (current BASIC statement execution is not completed before transferring). For these reasons all assembly language interrupts must have a higher priority than those handled by POWER BASIC; it is acceptable for an assembly language processor to interrupt a POWER BASIC interrupt processor but the reverse should never be allowed to occur.

Since assembly language interrupts are processed immediately and the POWER BASIC environment prior to the interrupt is not saved, it is not advisable to use the Floating Point XOPS of POWER BASIC in the assembly language processor.

TABLE 5-3

INTERRUPT LEVEL DATA

Interrupt Level	Vector Location (Memory Address In Hex)	Device Assignment	Interrupt Mask Values to Enable Respective Interrupts (ST12 thru ST15)
(Highest priority) 0	00	Reset	0 through F*
1	04	External device	1 through F
2	08	External device	2 through F
3	0C	Clock	3 through F
4	10	External device	4 through F
5	14	External device	5 through F
6	18	External device	6 through F
7	1C	External device	7 through F
8	20	External device	8 through F
9	24	External device	9 through F
10	28	External device	A through F
11	2C	External device	B through F
12	30	External device	C through F
13	34	External device	D through F
14	38	External device	E and F
(Lowest Priority) 15	3C	External device	F only

*Level 0 can not be disabled.

5.10 BASE STATEMENT

The BASE statement sets the CRU base address for subsequent CRU operations.

Form:

[line number] BASE <expression>

The BASE Statement evaluates the expression and sets the CRU base address to the result for use by teh CRB and CRF functions. The CRB function addresses bits within +127 and -128 of the evaluated base address. The CRF function transfers bits using the evaluated base address as the starting CRU address.

The CRU provides a maximum of 4096 input and output lines that may be individually selected by a 12-bit address. The 12-bit address used by the CRU instructions is actually located in bits 3 through 14 of a workspace register. The evaluated expression of the BASE statement is loaded into the entire 16-bits of this workspace register. Therefore, the BASE expression should evaluate to twice the actual (physical) CRU base address desired since only bits 3 throgh 14 are used. The least significant bit of the BASE expression value is ignored for CRU operations. Therefore, all expressions should evaluate to an even number. The range of valid expressions is from 0 to 8190 (hexadecimal 1FFE).

Examples:

```
10   BASE 64
20   CRF(0)=-1
30   BASE 100
40   CRB(-1)=0
```

Statement 10 sets the CRU BASE address to 64 (physical address of 32), and statement 20 outputs a 16-bit -1 value. Statement 30 sets the CRU BASE address to 100 (physical address of 50, and statement 40 sets the CRU bit displaced -1 from the base (physical address of 49) to zero.

5.11 TIME STATEMENT

The TIME statement is used to display, store, or set the 24 hour time of day clock.

Forms:

[line number] TIME

[line number] TIME <string variable>

[line number] TIME <exp>,<exp>,<exp>

The time of day may be directly displayed at any point within the program. It may also be displayed from the keyboard when in idle mode by using the first form of the TIME statement. The time of day will be displayed in the following format:

HH:MM:SS

Examples:

```
     TIME  9:31:23  (in keyboard mode)
10   TIME  11,4,0
```

```
          .
          .
          .
100    TIME
110    STOP

RUN
 11:04:37

 STOP AT 110
```

The second form of the TIME statement enables the current time of day to be stored in a string variable. This is useful for recording occurrence time of significant events in a user's application program.

Example:

```
10     DIM  T(3)
20     TIME 11,4,0
          .
          .
          .
100    TIME  $T(0)
120    PRINT $T(0)
130    STOP

RUN
 11:04:37
 STOP AT 130
```

The TIME statement is one of the POWER BASIC statements which functions slightly different on the Host POWER BASIC Interpreter than on the Target POWER BASIC Interpreter. The difference between the two systems in the operation of the TIME statement lies solely in the setting of the time. These differences will be presented in the appropriate paragraphs below.

Host POWER BASIC Interpreter:

Setting of the time via the "TIME ⟨exp⟩,⟨exp⟩,⟨exp⟩" statement is not supported on the Host POWER BASIC Interpreter. Instead the user will set the time by using the INITIALIZE DATE AND TIME (ID) command of the SYSUTL program module. The user must set the time and date (if TIME is desired during BASIC program developemnt) before executing the Host POWER BASIC Interpreter.

To set the time, the user must load SYSUTL using either OCP or the TXDS Control Program by performing the steps in "Diskette OCP System Utility (SYSUTL) Program" (Section VIII of the TX990 OPERATING SYSTEM

PROGRAMMER'S GUIDE). In response to the "OP:" prompt, the user will enter the ID command. The syntax of the command is as follows:

ID, <year>,<month>,<day>,<hour>,<minute>.

The year operand is the four-digit decimal number of the years 1976 through 1999, and the month operand is the decimal number of the month 1 through 12. The day operand is a one- or two-digit decimal number, 1 through 31, and the hour operand is a one- or two-digit decimal number, 0 through 23. The minute is the decimal number of the minute 0 through 59. The second is set to zero when the command is entered.

Example:

OP: ID,1978,11,15,15,28.

15:28:00 NOV 15, 1978

Note that the "TIME <exp>,<exp>,<exp>" statement will be accepted by the Host POWER BASIC Interpreter and will be checked only for statement syntax. Execution will continue with the next POWER BASIC statement. Therefore the user may insert the appropriate "TIME <exp>, <exp>,<exp>" statement into the application program, it will have no effect in Host POWER BASIC Interpreter execution, and it will be appropriately configured into the Target POWER BASIC Interpreter. The user will enter the "TIME <exp>,<exp>,<exp>" statement into the application program according to the explanation below on the Target POWER BASIC Interpreter.

Target POWER BASIC Interpreter:

The "TIME <exp>,<exp>,<exp>" statement is used to start and set the time of day clock of the Target POWER BASIC Interpreter on the TM990 board system. the form of the expressions is as follows:

TIME HH,MM,SS

where

H = hours, M = minutes, S = seconds

The clock of the Target POWER BASIC Interpreter is set up as a 24-hour clock with times ranging from 00:00:00 to 23:49:59. Initialization of the clock is valid at any point wihtin the application program. Its value may be reinitalized at any point.

Examples:

```
10 TIME 9,0,0
10 TIME 12,30,0
```

To directly set the time within a configured application program as illustrated in the above examples, requires that the POWER BASIC system to be initialized at the exact time specified within the program in order for the TIME statement to be accurate. The typical method of setting the time in a configured application program of the Target POWER BASIC Interpreter is illustrated in the following program.

Example:

```
10  PRINT "INITIALIZE TIME-OF-DAY CLOCK"
20  INPUT "HOURS = ";HRS, "MINUTES = ";MIN,"SECONDS = ";SEC
30  TIME HRS,MIN,SEC
```

5.12 RANDOM STATEMENT

The RANDOM statement randomizes the seed for the pseudo-random number generator.

Forms:

[line number] RANDOM <expression>

The RANDOM statement is used in conjunction with the RND function. The RND function returns the next number in the random number sequence. It returns this value when requested and replaces it with the next random number. The RANDOM statement is used to change the random number seed and therefore the sequence of pseudo-random numbers.

The random seed is set to a constant value when POWER BASIC is first initialized so that the RND variable will always return the same sequence of numbers to facilitate program debugging. After the debugging phase, the RANDOM statement may be used to alter this sequence.

The RANDOM statement is used to set the seed to a specific or arbitrary value. The expression is evaluated and the result used as the seed of the random number generator. The expression may be any valid POWER BASIC expression. The evaluated expression must be within the limits of -32768 and 32767 or a FIX ERROR will result. The sequence of numbers generated by a specific seed value will always be the same. This is useful for debugging and testing an application program with a predetermined seed value. Arbitrary seed values may be generated by the user by using combinations of variables and functions (including the RND function) within the expression.

Examples:

```
10  RANDOM 220
20  RANDOM 1000*RND
30  RANDOM RND*MEM(X)
```

5.13 ESCAPE AND NOESCAPE STATEMENTS

The ESCAPE and NOESC statements provide the capability to enable or disable the escape key to interrupt program execution on the Host POWER BASIC Interpreter. These statements are not supported by the Target POWER BASIC Interpreter and therefore function as a STOP statement if encountered within a application program executing on the Target BASIC Interpreter.

```
[line number] ESCAPE
[line number] NOESC
```

The ESCAPE statement enables the terminal device escape (or break) key to interrupt program execution. The escape key is normally enabled in POWER BASIC unless an NOESC statement has been executed. When the escape key is struck, the program terminates upon completion of the current statement line. Keyboard sampling during the RUN mode is performed only between statement lines. Caution should be observed when certain statement constructions are used. For example, the FOR and NEXT statements should not appear in the same statement line, because a statement line is autonomous. Once the FOR/NEXT line begins execution, it cannot be interrupted by using the escape key. It can be interrupted only if the end condition of the FOR/NEXT loop is met, or if the user reinitializes the system via the reset swtich on the CPU board.

The NOESC statement disables the terminal device escape (or break) key from interrupting program execution.

The ESCAPE statement (default condition) is used during program development and debug. The NOESC statement is used for time critical application programs or in an environment where it is not desirable for the user to interact with POWER BASIC in a non-program controlled mode.

Examples:

```
10 ESCAPE
10 NOESC
```

Target POWER BASIC Interpreter: The ESCAPE and NOESC statements are not supported by the POWER BASIC Configurator, therefore these statements should not be present in a final application program which is to be configured into a customized (Target) POWER BASIC Interpreter. The statements will function as a STOP statement if encountered in a Target application program.

5.14 CALL STATEMENT

The CALL statement allows the user access to assembly language subroutines. The user may pass up to 4 patameters to the subroutine although none are required.

Forms:

[line number] CALL <string constant>,<address><,var1><,var2><,var3><,var4>

where:

string constant is the entry point of the assembly language subroutine

address is the hexadecimal or decimal address of the assembly language subroutine

var1, var2, var3, and var4 are the parameters of the subroutine

The string constant is the entry point of the assembly language subroutine being called (used by the POWER BASIC Configurator and the Target POWER BASIC Interpreter). The address is the decimal or hexadecimal address where the asembly language subroutine resides (used by the Host POWER BASIC Interpreter). Both the string constant and the address parameters must be entered to execute the CALL statement, although only one is used by each of the two POWER BASIC Interpreters.

Parameters are passed to the assembly language subroutine via the variables <var1> through <var4>. Up to four parameters may be passed between the POWER BASIC program and the subroutine, although none are required. Parameters may be passed by either value (ie., the integer value of the variable in the POWER BASIC program) or by address (ie., the address where the variable is stored in POWER BASIC). A variable is passed by value when the variable itself is entered as a parameter (eg., ABC, VAR, DON, etc.), and the address of the variable is passed when the variable is enclosed in parenthesis (eg., (ABC), (VAR), (DON), etc.). If the parameter is passed as a value, it will be converted into a 16-bit two's complement integer. If passed by address, the 16-bit address of the location in memory where the variable is stored internal to POWER BASIC will be passed, and no variable conversion will be performed. Parameter passing by address

enables the user to access both floating point and dimensioned or array variables from the assembly language subroutine, knowing that each variable or array element is 6 bytes in length. The user should refer to Section 3, paragraph 3.7.6 for detailed information on the internal integer and floating point variable formats used in POWER BASIC.

The CALL statement internally performs a "BL" assembly instruction to access the users assembly language subroutine. The parameters of the CALL statement are passed in Registers 4, 5, 6, and 7 of the POWER BASIC workspace prior to execution of the "BL " instruction. The user may then directly access these parameters from his assembly language subroutine. The return address is contained in R11 of the POWER BASIC workspace.

Since a "BL" instruction is used, the assembly language subroutine may use only registers 4, 5, 6, and 7 of the POWER BASIC workspace. If other registers than these are modified within the assembly language program, values expected by POWER BASIC will be destroyed, and unpredictable results may occur when returning from the assembly language routine. Therefore it is suggested, that the assembly language routines supply their own workspaces. Typically, this is done by performing the following sequence immediately upon entering the assembly language subroutine:

```
*
*  ENTRY POINT TO THE ASSEMBLY LANGUAGE SUBROUTINE
*
SUBR1  EQU   $
       BLWP  @SUBR1Z          GET OWN WORKSPACE
       B     *R11             RETURN TO POWER BASIC
*
*  TRANSFER VECTOR.
*
SUBR1Z EQU   $
       DATA  WRPTR            WORKSPACE POINTER
       DATA  START            PROGRAM COUNTER
*
*  START OF ASSEMBLY LANGUAGE SUBROUTINE
*
START  EQU   $
         .
         .
         .
         .
         .
       RTWP                   RETURN FROM ASSEMBLY SUBROUTINE
```

The CALL statement is one of the statements having the same form, but its function is slightly different on the Host POWER BASIC

Interpreter, than in an application program configured into the Target POWER BASIC Interpreter. The address parameter is used by the Host POWER BASIC Interpreter to access an assembly language routine which has been loaded into a specified memory location, while the string constant parameter is used by the POWER BASIC Configurator and TX990 Link Editor to appropriately link the assembly language subroutine into the customized Target POWER BASIC Interpreter. The differences in the use of the CALL statement between these two POWER BASIC Interpreters will be presented below.

Host POWER BASIC Interpreter:

To access an assembly language subroutine from the Host POWER BASIC Interpreter, the user must load the object program from either cassette or a diskette file. To load an object module using POWER BASIC, the user must load and execute the POWER BASIC Object Loader. This program is presented in the POWER BASIC Object Loader Application Note available from your distributor. This BASIC program will appropriately load an object file from either cassette or diskette beginning at the specified load point entered by the user.

When the Host POWER BASIC Interpreter is initially executed, it automatically sizes the contiguous RAM area from the top of Interpreter up to memory address hex F800 and allocates all available RAM area to the BASIC Interpreter for variable and BASIC program storage. However, a small area of RAM must be available for the object program and its associated workspace. To provide this area, the user must execute the "NEW address " command immediately after executing the POWER BASIC Interpreter before loading any POWER BASIC programs. The address will specify the upper memory limit to be used by the POWER BASIC Interpreter. The user may then load the object program into RAM at an address above this limit. The user may then execute a CALL statement to this address to execute his assembly language subroutine.

Target POWER BASIC Interpreter:

Any assembly language subroutines CALLed by a BASIC application program (which is to be configured into a Target POWER BASIC Interpreter) will be included into the final Target BASIC Interpreter by the configuration process. When the POWER BASIC Configurator encounters a CALL statement during its scan of the application program, it will interpret the string constant as the FILE NAME and ENTRY POINT of the object module being referenced, and will issue an "INCLUDE" statement for that object module in the Link Control File which it produces. The string constant may consist of up to six alphanumeric characters, the first of which must be alphabetic. This name must correspond to both the ENTRY POINT and FILE NAME of the object module being referenced for correct inclusion by the

Configurator and Link Editor. The Format of the INCLUDE statement will be as follows for each object module CALLed by the BASIC application program:

```
INCLUDE  DSC2:XXXXXX/OBJ
```

where:

XXXXXX is the string constant used in the CALL statement

The TX990 Link Editor will then appropriately include the specified object modules from diskette, and the assembly language subroutines will be linked with the final Target POWER BASIC Interpreter object module.

> Note that the user must have the object module located on the diskette under the specified file name (with the "/OBJ" extension), and this diskette MUST be located in disk drive 2 of the FS990 system before Link Editor is called into execution.

The user must separate his workspace and data areas from the body of all assembly language subroutines since all assembly language subroutines will be included in the final Target POWER BASIC Interpreter, and this final object module will be placed in ROM for execution in a TM990 board system. Therefore, a small amount of RAM area will need to be allocated for the workspaces and data areas in the final TM990 target system. The Target POWER BASIC Interpreter during Power-up RESET will size for all available RAM area in the TM990 board system, starting from memory address hex $FFFF_{16}$ down. All available contiguous RAM area will then be assigned to the BASIC Interpreter for variable storage. Therefore the user must supply a non-contiguous block of RAM in the TM990 board system for use by the assembly language subroutines.

Examples:

```
200 CALL  "CONVERT",0B800H,VAL,UPB,LWP
200 CALL  "INOUT",0C800H,ST,(VAR),(NAM)
```

The first example will branch and link (BL) to the subroutine "CONVERT" at location hex B800 with the 16-bit two's complement values of the variables VAL, UPB, and LWB passed in registers 4, 5, and 6. If executed on the Host POWER BASIC Interpeter, the assembly language subroutine must be loaded so that its entry point is at hex address $B800_{16}$. If configured, the corresponding object module should be located in the file "DSC2:CONVERT/OBJ" to be correctly included by the Link Editor. The second example will branch and link to the subroutine "INOUT" at location hex $C800_{16}$. The 16-bit two's complement value of ST will be passed in Register 4, while the memory address of the location of the variables VAR and NAM will be passed in

registers 5 and 6, respectively.

5.15 FILE MANAGEMENT

The Host POWER BASIC Interpreter provides a convenient method of performing file I/O to the devices and diskette files of the TX990 Operating system via its file management package. The Host POWER BASIC Interpeter supports a minimal but sufficient set of file primitives to SAVE and LOAD BASIC programs, and read and write binary data. The SAVE and LOAD Commands were discussed in Section 4, while the remainder of the file primitives will be discussed below.

5.15.1 PATHNAME SYNTAX

Most all POWER BASIC file control statements use a pathname to indicate a device/file referenced by that statement. All TX990 pathnames used by POWER BASIC adhere to the following rules:

- All device pathnames consist of a one- to four- character device name assigned to that device during system generation. Typical device pathnames are DSC, DSC2, LP, CS1, and CS2. The user must reference his system generation for the device names used in his particular TX990 Operating System.

- All diskette file pathnames consist of one to seven characters separated from the device name by a colon (:). The first character must be alphabetic (A-Z); the rest may be alphanumeric. The file name is followed by an extension to the file name, which is one to three characters separated from the file name by a slash(/). The first character must be alphabetic; the rest must be alphanumeric. The file name and extension are specified when the file is created by the BDEFS or BDEFR POWER BASIC statements. Typical valid diskette file names are DSC2:LOAD/ONE, DSC:PROGRAM/T1A, and :TEMPER/D11. Invalid diskette file names would be DSC:TEMPERATURE/SNC -- too many characters, DSC2:FANON/920 -- first character of extension must be alphabetic, and DSC:MOTOR/STEP -- extension too long.

The POWER BASIC/TX990 interface checks the syntax of all file and device pathnames when the BASIC statements are entered. If the pathname syntax is not legal, an INVALID PATHNAME error will result.

5.15.2 BDEFS STATEMENT

The BDEFS statement will define a sequential diskette file on the specified device and with the specified file name and extension. The

BDEFS file primitive is a POWER BASIC statement; however, it is typically only executed in the keyboard mode (without a statement number), and is not entered into a BASIC program since each execution of the program would attempt to redefine the file. Attempts to define a file which already exists will result in a TX990 error.

Form:

BDEFS { <string constant> / <string variable> }

The <string constant> or <string variable> specifies the pathname of the diskette file to be defined.

The file is placed in the diskette file directory, but the file is not "known" by POWER BASIC until the file is opened by the BOPEN statement.

The logical records in a sequential file must be accessed in a sequential manner (ie., record 1 must be processed before record 2, etc). When a sequential file is closed (via the BCLOSE statement following an access), the position of the last access to the file is saved. When the file is open again (via the BOPEN statement), the next I/O operation accesses the next logical record in the file, instead of the first record of the file. To access the first record of the file, the user must execute the RESTOR #<var> statement to rewind the file to its beginning.

Sequential files in POWER BASIC are exclusively opened for read only or write only operations. The access type is specified by the first read or write to that device/file, and thereafter the file is defined as either a read only or write only LUNO. The user cannot write to a file and then attempt to read values from the file without first closing the file after the write via the BCLOSE statement, and vice versa. Attempts to read a sequential file opened for write or to write a sequential file opened for read will result in a INVALID FILE ACCESS error message. Also note that the last write operation to a sequential file defines the current end of file.

Examples:

```
BDEFS  "DSC2:RELAY/SEQ"
BDEFS  "DSC:MOTOR/CNT"
BDEFS  "DSC:VALUE/OPN"
```

5.15.3 BDEFR STATEMENT

The BDEFR statement will define a relative record (random access) diskette file on the specified device and with the specified file name and extension. The BDEFR file primitive is a POWER BASIC statement; however, it is typically only executed in the keyboard mode (without a

statement number) and is not entered into a BASIC program since each execution of the program will attempt to redefine the file. Attempts to define a file which already exists will result in a TX990 error.

Form:

BDEFR { <string constant> / <string variable> }

The <string constant> or <string variable> specifies the pathname of the diskette file to be defined.

The file is placed in the diskette file directory; however, the file is not "known" to POWER BASIC until the file is opened by the BOPEN statement.

Relative record files are supported only on diskette. The beginning of a relative record file is specified by logical record 0; the end of the relative record file is specified by the highest numbered logical record written to the file. When relative record files are opened, they may be accessed for both read and write operations. That is, the user may read a record and then immediately write a record without performing a close operation between them.

When a relative record file is opened via the BOPEN statement, the file is positioned to its beginning (logical record number of 0). The relative record file may then be written or read in a sequential manner until the highest record number is reached. After a read or write operation, the logical record number is appropriately incremented by the number of bytes read or written. The user may also access any record at random within the file by using the BINARY 4 statement. The user will specify the record length, logical record number, and byte displacement within the record when entering the BINARY 4 statement. The file will then be positioned to that particular record and byte within the file. The next read or write may then access this "random" position within the file. Subsequent accesses may be either random or sequential. When only BINARY 1, BINARY 2, or BINARY 3 statements are executed, the next record in the sequence is accessed. When another BINARY 4 statement is executed with a position value out of sequence, the access is at "random".

Examples:

```
BDEFR  "DSC2:GATE/REL"
BDEFR  "DSC:SWITCH/ACT"
BDEFR  "DSC:TIME/MON"
```

5.15.4 BDEL STATEMENT

The BDEL statement will delete the specified sequential or relative record file from the specified diskette. The BDEL file primitive is a POWER BASIC statement; however, it is typically only executed in the keyboard mode (without a statement number), and is not entered into a BASIC program since each execution of the program will attempt to delete the specified file. Attempts to delete a file which does not exist will result in a TX990 error.

Form:

BDEL { <string constant> / <string variable> }

The <string constant> or <string variable> specifies the pathname of the diskette file to be deleted.

The BDEL statement deletes the file name from the diskette file directory, and subsequent references to this file name will result in a undefined file name error. The specified file must be closed before deletion. Attempts to delete a file which is opened will result in a TX990 error.

Examples:

```
BDEL  "DSC:MOTOR/CNT"
BDEL  "DSC2:GATE/REL"
```

5.15.5 BOPEN STATEMENT

The BOPEN statement will open the specified device/file for sequential or relative record access depending upon the device/file type. Most devices (such as the 733 ASR/KSR keyboard/Printer, line printer, and 733 ASR Cassette Unit) may only be opened for sequential access, while diskette files will be opened according to the file type as defined by the BDEFS or BDEFR statements. Note that if a diskette file is specified as the pathname of the BOPEN statement, the file must have previously been defined by the BDEFS on BDEFR statements.

The BOPEN statement will place the specified file name in the operating system and in the POWER BASIC file directory. Until an open operation is executed, devices/files are not located and file management operations to manipulate them cannot be initiated or performed. The BOPEN statement causes the device/file to be assigned solely to the calling program until a BCLOSE statement is executed on the device/file. That is, no other BASIC program may access the device/file until it is properly closed.

Form:

[line number] BOPEN {<string constant> / <string variable>}, <numeric variable>

The <string constant> or <string variable> specifies the pathname of the device/file to be opened. A logical unit number (LUNO) is assigned to the device/file name when it is placed into the POWER BASIC directory. This LUNO is then returned to the user in the <numeric variable>. The numeric variable (LUNO) is used in all future file management operations when referencing the the device/file (e.g., the COPY <numeric variable> TO <numeric variable> and BCLOSE <numeric variable>).

The file is opened for either read and/or write. A relative record file will be open for both read and write; that is the file may be both written to and read from without closing the file between operations. However, a sequential file will be opened exclusively for read only or for write only depending on the type of the first access to the file after it is opened.

Note that user may only have up to 4 device/file LUNO's opened at one time. If more than 4 devices/files are attempted to be opened, a TABLE AREA FULL error will result.

The BOPEN statement performs only an open operation. That is, it does not rewind the device/file to its beginning before returning to the user.

A relative record file is still positioned to the first logical record of the file.

A sequential diskette file, when closed following an access, saves the position of the last access to the file. When the file is opened again, the next I/O transfer accesses the next logical record in the file instead of the first logical record of the file. To access the first logical record after a BOPEN operation, the user must execute a rewind operation (via the RESTOR #<var> statement). Typically, when entering values into a new file, or reading or overwriting an existing file, the user executes a RESTOR statement immediately after opening the file and before performing any I/O operations to that diskette file so that subsequent accesses will begin with the first record of the file.

If the specified device/file is the 733 ASR cassette unit, the user will need to perform a RESTOR operation on the cassette device if the first record of the tape is to be read from or written to.

The BOPEN statement is typically used in both the keyboard mode (without a statement number) and in POWER BASIC programs. In the

keyboard mode it is typically used to open the devices/files used by the COPY statement; while in a program, it opens the devices/files used in BINARY statements.

Examples:

```
      BOPEN "DSC2:FILE1/SRC",FIL
      RESTOR #FIL
      BOPEN "LP",LP
      COPY  FILE TO LP
   50 DIM A(5)
   60 $A(0)="DSC:FAN/TST"
   70 BOPEN $A(0),FAN
   80 RESTOR #FAN
   90 BOPEN "CS1",CSO
  100 BINARY 1,FAN,80;3,VAL(0)
```

5.15.6 BCLOSE STATEMENT

The BCLOSE statement will close the specified device/file. The BCLOSE statement removes the device/file from the POWER BASIC directory, releases the assigned logical unit number (LUNO), and releases the I/O device and the file on the medium of the I/O device from the POWER BASIC Interpreter.

Form:

[line number] BCLOSE <numeric variable>

The numeric variable specifies the logical unit number (LUNO) that was assigned to it by the corresponding BOPEN statement.

The BCLOSE statement will perform a straight close operation if the specified file was opened and read from, but not written to. The close operation is as explained above.

The BCLOSE statement will perform a close with end-of-file (EOF) operation if the file was opened and written to. It will perform a close operation as previously described above, followed by a write EOF operation. When the device specified is the keyboard/printer, the printer performs three line feed operations. When the device specified is the 911 or 913 VDT, only the close operation is performed. When the device specified is the line printer, the printer performs a form-feed operation. When the device/file specified is a sequential diskette or cassette file, the EOF sequence is output at the current position of the file. When the device/file is a relative record diskette file, the EOF sequence is output after the last record of the file.

The BCLOSE statement is typically executed in both the keyb
and in POWER BASIC programs. In the keyboard mode it is
used to close the devices/files used in a COPY statement; w
program, it is used to close the devices/files used by th
statement.

Examples:

```
        BOPEN "DSC:MOTOR/SRC",MOT
        RESTOR #MOT
        BOPEN "CS1",CSO
        RESTOR #CSO
        COPY  MOT TO CSO
        BCLOSE  MOT
        BCLOSE  CSO
 10     BOPEN "DSC2:DATA/BIN",DAT
 20     RESTOR #DAT
 30     BINARY 1,DAT,80;3,VAL(0)
 40     BCLOSE  DAT
 70     BOPEN "LP",I
 90     BCLOSE I
```

5.15.7 RESET STATEMENT

The RESET statement will close ALL open devices/files
device/file names will be removed from the POWER BASIC directo
logical unit numbers (LUNO's) will be released, and the device
will be released from the POWER BASIC Interpreter. The
statement has no parameters.

Form:

```
[line number]  RESET
```

The RESET statement will perform a straight close operation
devices/files which were opened and only read from, and not w
to.

The RESET statement will perform a close with end-of-file
operation to all devices/files which were open and written to.
to the BCLOSE statement for an explanation on the close wi
operation.

Examples:

```
BOPEN "DSC2:VALUE/FIN",SRC
RESTOR #SRC
```

```
        BOPEN "DSC:VALUE/FIN",DST
        RESTOR #DST
        COPY  SRC TO DST
        RESET
10      BOPEN "DSC2:PARM/REL",PAR
20      RESTOR #PAR
30      BINARY 1,PAR,16;2,NUM(0)
60      BOPEN "LP",J
90      RESET
```

5.15.8 COPY STATEMENT

The COPY statement allows the user to copy one file on a specified device to another device/file. The COPY statement is useful for backing-up, expanding, printing, or concatenating cassette or diskette files.

Form:

[line number] COPY <numeric variable 1> TO <numeric variable 2>

The file specified by <numeric variable 1> is copied into the file specified by <numeric variable 2>. The <numeric variable> specifies the logical unit number (LUNO) that was assigned to a particular device/file by the corresponding BOPEN statement. This format requires that both files must be defined and opened prior to execution of the copy statement.

When copying files, both devices/files must be of the same file type, tht is, both sequential or both relative record. If a sequential file is attempted to be copied to a relative record file, or visa versa, an INCOMPATABLE FILE TYPE error will result.

Examples:

```
BOPEN "DSC2:PROCESS/SR1",S1
RESTOR #S1
BOPEN "DSC2:PROCESS/SR2",S2
RESTOR #S2
BOPEN "DSC2:PROCESS/SRC",SRC
RESTOR #SRC
COPY  S1 TO SRC          !COPY PART 1
COPY  S2 TO SRC          !APPEND PART 2 - SAVE PROGRAM IN SRC
RESET
BOPEN "DSC2:PROCESS/SRC",SRC
RESTOR #SRC
BOPEN "DSC:PROCESS/BCK",BCK
RESTOR #BCK
COPY  SRC  TO  BCK       !BACKUP PROGRAM
RESET
```

```
 10  BOPEN "DSC2:SAMPLES/TST",SMP
      .
      .
 70  RESTOR #SMP                    !REWIND SAMPLE FILE TO B
 80  BOPEN "LP",LP
 90  COPY SMP TO LP                 !PRINT SAMPLE FILE TO LP
100  RESET
110  STOP
```

The COPY statement may also be used to expand sequent: files as follows:

```
      BDEFS       DSC2:PROCESS/TMP
10    REM - EXPAND THE DISKETTE FILE DSC2:PROCESS/TST
20    BOPEN  "DSC2: PROCESS/TST", TST
30    RESTOR # TST
40    BOPEN "DSC2:PROCESS/TMP", TMP
50    RESTOR # TMP
60    COPY TST TO TMP
70    BCLOSE TST
80    REM - ADD NEW RECORDS TO FILE DSC2: PROCESS/TMP
90    BINARY 1, TMP, 80; 2, VAL (0)
       .
       .
       .
       .
       .
       .
       .
```

5.15.9 BINARY DATA I/O STATEMENTS

The BINARY I/O statements permit reading and writing of b values from the specified device/file. The user may read by from the device/file into successive variables, or may values from successive variables expressions to the s device/file. The byte values are read or written as 8-l values with no conversion. This enables storing and accessi values on the specified device/file.

File Record Length = Fixed 128

There are four forms of the BINARY statement. They are used to specify the number of bytes to be transferred (BINARY 1), write an assigned number of bytes from each succeeding expression to specified device/file (BINARY 2), read an assigned number of bytes from specified device/file into each succeeding variable (BINARY 3), or specify the position within a relative record file (BINARY 4).

The general form of the BINARY statement is as follows:

[line number] BINARY <exp>,<arg>,<arg>,<arg>

where

<exp> specifies the type of BINARY operation to be performed
<arg> are the arguments of the particular BINARY operation

The four forms of the BINARY statement will be explained in detail in the following sections.

Several BINARY statements may be concatenated into a single BINARY statement on one line by separating all succeeding BINARY statement <expressions> and <arguments> from the preceeding BINARY statement with a semicolon (;). Note that the "BINARY" statement name will appear only as the first statement of the line, and will not appear before subsequent BINARY statement <expressions> and <arguments>.

Example:

```
10  BOPEN "DSC:FILE/SRC",FIL
20  RESTOR #FIL
30  BINARY 1,FIL,6
40  BINARY 2,A,B
50  BINARY 3,VAL,VAR
```

could be concatenated into a single statement as

```
10  BOPEN "DSC:FILE/SRC",FIL
20  RESTOR #FIL
30  BINARY 1,FIL,6;2,A,B;3,VAL,VAR
```

Concatenation of BINARY statements is required when more or less than 6 bytes are to be read or written from the device/file. This is required since execution of the "BINARY" phrase (of any subsequent BINARY 2, 3, or 4 statements) resets the number of bytes to be read or written to 6, the typical default byte length since all POWER BASIC variables and array elements are 6 bytes in length.

Example:

```
10  BOPEN "DSC2:FILE1/SRC",FIL
20  RESTOR #FIL
30  BINARY 1,FIL,80                 !SET # OF BYTES TO 80
40  BINARY 3,REC(0)                 !***WILL READ ONLY 6 BYT
50  BINARY 2,OUT(0)                 !***WILL WRITE ONLY 6 BY
```

the correct form would be

```
10  BOPEN "DSC2:FILE1/SRC",FIL
20  RESTOR #FIL
30  BINARY 1,FIL,80;3,REC(0);2,OUT(0) !WILL CORRECTLY READ &
```

5.15.9.1 BINARY 1 STATEMENT

The BINARY 1 statement specifies the logical unit number (L the number of bytes per device/file access to read from or wr that LUNO for all subsequent BINARY statements.

Form:

[line number] BINARY 1, <numeric variable>, <exp>

The <numeric variable> specifies the logical unit number (LUNO device/file to which all subsequent BINARY operations a performed. The LUNO is the number which is assigned to the s variable when the device/file is opened with the BOPEN st After executing the BOPEN statement, the specified variable (c represents the device/file pathname, and this variable is use BINARY statement. The BINARY 1 statement assigns the BINARY L use by all subsequent BINARY statements.

The <exp> specifies the number of bytes to be read from or wri the BINARY LUNO for each subsequent access by the BINARY 2 or statements. Note that the scope of the <exp> is only for the BINARY statement. Execution of the next BINARY statement wi the number of bytes to 6. Therefore in applications where less than 6 bytes are to be read or written for each access, must concatenate the BINARY 2 or BINARY 3 statements to the B statement via the semicolon (;) separator.

Example:

```
10  DIM A(10)
20  BOPEN "DSC:MOTOR/CNT",MTR
30  RESTOR #MTR
30  BINARY 1,MTR,66;2,A(0)          !WRITES 66 BYTES TO FI
```

while,

```
10  DIM A(10)
20  BOPEN "DSC:MOTOR/CNT",MTR
30  RESTOR #MTR
40  BINARY 1,MTR,66 :: BINARY 2,A(0) !ONLY WRITES 6 BYTES TO FILE
```

Example:

```
10  BOPEN "DSC2:TEMPER/SMP",TEM
20  RESTOR #TEM
30  BINARY 1,TEM,6                    !SPECIFY FILE FOR 6 BYTES/ACCESS
40  BINARY 3,TM1,TM2,TM3,TM4          !READ FOUR 6-BYTE DATA VALUES
```

5.15.9.2 BINARY 2 STATEMENT

The BINARY 2 statement writes an assigned number of bytes from each succeeding expression to the BINARY LUNO. The BINARY LUNO and number of bytes are assigned by the BINARY 1 statement.

Form:

[line number] BINARY 2, <exp> [,exp] ....

The <exp> of the BINARY 2 statement specifies the expression or variable values to be written to the BINARY LUNO.

The BINARY LUNO is assigned by the most recently executed BINARY 1 statement. The number of bytes to be written is also assigned by the BINARY 1 statement if the BINARY 2 statement is concatenated with the BINARY 1 statement via the semicolon separator (eg., BINARY 1,FIL,20;2,VL1(0),L2(0) will write 20 bytes from each of the variables VL1(0) and L2(0)). If the BINARY 2 statement is a separate statement entry, the number of bytes to be written is reset to 6 (eg., BINARY 2,VL1,VL2 will write 6 bytes from the variables VL1 and VL2).

Example:

```
10  DIM A(10)
20  BOPEN "DSC:STRING/CHR",CHR
30  RESTOR #CHR
40  $A(0)="STORE THIS CHARACTER STRING INTO FILE"
40  BINARY 1,CHR,66;2,A(0)        !OUTPUT STRING
```

This example will store the character string $A(0) into the file "DSC:STRING/CHR". Note that string variables are placed in the BINARY statements as numeric variables, not string variables. This is because all variables are output directly without regard to either numeric or ASCII content.

Example:

```
10  BOPEN "DSC2:DATA/SQR",SQR
20  RESTOR #SQR
30  BINARY 1,SQR,6
40  BINARY 2,I,I*I,SQR(I)
50  NEXT I
```

This example will output the values I, I*I, and SQR(I) to "DSC2:DATA/SQR" for the values of I ranging from 1 through 10

5.15.9.3 BINARY 3 STATEMENT

The BINARY 3 statement reads from the BINARY LUNO an assign of bytes into each of the succeeding variables. The BINARY number of bytes are assigned by the BINARY 1 statement.

Form:

[line number] BINARY 3; <variable> <,variable> ....

The <variable> of the BINARY 3 statement specifies the vari receive the values read form the BINARY LUNO. They may b simple or dimensioned numeric variables.

The BINARY LUNO is assigned by the most recently executed statement. The number of bytes to be read is also assigned BINARY 1 statement if the BINARY 3 statement is concatenated BINARY 1 statement via the semicolon separator (eg., 1,FIL,40;3,VAL(0) will read 40 bytes into the variable VAL(the BINARY 2 statement is a separate statement entry, the nu bytes to be read is reset to 6 (eg., BINARY 3; VL1,VL2 wil bytes into each of the variables VL1 and VL2).

Examples:

```
10  DIM CHR(4),SET(4)
20  BOPEN "DSC2:CHAR/SET",FL1  !OPEN THE SEQUENTIAL FILE
30  RESTOR #FL1
40  $CHR(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
50  BINARY 1,FL1,30;2,CHR(0)    !WRITE 30 BYTE VALUES FROM
60  BCLOSE FL1                   !CLOSE FILE TO WRITE EOF
70  BOPEN "DSC2:CHAR/SET",FL2  !OPEN FILE
80  RESTOR #FL2
90  BINARY 1,FL2,30;3,SET(0)    !READ 30 BYTES VALUES INTO
100 PRINT $SET(0)
110 RESET
120 STOP
```

Statement 50 of this program writes the character set in $CHR(0) to the file "DSC2:CHAR/SET". Note that the string variable ($CHR) is specified as a numeric variable (CHR) when referenced in the BINARY statement. It writes 30 characters, including 26 characters of the character set plus the terminating null. Since this is a sequential file, and it has been written to, the file must be closed before it can be read from. Statement 60 performs the close operation, while statement 70 opens the file so the data values may be read. Statement 90 reads the 30 characters saved in the file into the numeric variable SET(0). Again a numeric variable is specified in the BINARY statement instead of a string variable. Statement 100 then prints this character set as a string, $SET(0).

5.15.9.4 BINARY 4 STATEMENT

The BINARY 4 statement allows the user to access a given byte position within a relative record diskette file. A file is defined to be random access by the BDEFR statement when the file is created. The BINARY LUNO is assigned by the BINARY 1 statement. The number of bytes specified in the BINARY 1 statement has no effect on the BINARY 4 statement.

Form:

[line number] BINARY 4 ,<exp1>,<exp2>,<exp3>

where

<exp1> specifies the record length
<exp2> specifies the record number
<exp3> specifies the byte displacement within the record

The BINARY LUNO is assigned by the most recently executed BINARY 1 statement.

The record length specified by <exp1> is the byte length as determined by the user. A record may consist of a single variable or collection of variables as required by the application.

The record number specified by <exp2> indicates the record displacement within the file. The valid range for <exp2> is from 0 through the last record of the file. Record number 0 is the first record of the file.

The byte displacement specified by <exp3>indicates the byte position within the record specified by exp2 . The valid range for<exp3> is from 0 through the record length specified by <exp1>. A byte displacement of 0 indicates the first byte of the record.

The specified final position within a relative record file ca
determined by:

POSITION = (RECORD LENGTH * RECORD NUMBER) + BYTE DISPLACEME

The BINARY 4 statement can be entered as a single statement, or ma
concatenated with other BINARY statements using the semico
separator.

The relative record file may be positioned past the end of file
then written into via the BINARY 2 command. This will appropriat
expand the relative record file and the last record written w
immediately preceed the current end of file. All records between
previous end-of-file and the current end-of-file, even if not writ
into via the BINARY 2 statement, will be included in the fil
Attempts to position past the end of file on a relative record f
and then read values by the BINARY 3 statement will result in an TX
error. Attempts to position a sequential file via the BINARY
statement will result in a POSITION ERROR message.

Examples:

```
BDEFR "DSC2:FILE1/REL"          !ENTERED IN KEYBOARD MODE

 10  BOPEN "DSC2:FILE1/REL",FIL
 20  RESTOR #FIL
 30  FOR I=1 TO 100
 40  BINARY 1,FIL,6;2,I,SQR(I),I*I,I 3
 50  NEXT I
 60  INPUT "RECORD NUMBER ";NUM
 70  BINARY 4,24,NUM,0;3,I,SQI,II,III
 80  PRINT I,SQI,II,III
 90  INPUT "ANOTHER RECORD? (Y or N)"; $M
100  IF $M = "Y" THEN GOTO 60
110  BCLOSE FIL
120  STOP
```

This example defines a relative record file "DSC2:FILE1/REL", and th
writes the values of I,SQR(I), I*I, and I 3 for I=1 to 100 into th
file, outputting 6 bytes for each value written. It then asks th
user for a particular record number within this file, and the
appropriately positions within the file and outputs the curren
record. Note that length specified in the BINARY 4 statement wa
taken to be 4*6 or 24, since this partitioned the file such that eac
record appropriately contained the values of I, SQR(I), I*I, and I 3
Also note that the byte displacement was taken to be 0 since th
values of interest started with the first byte of the records.

5.15.9.5 EXAMPLE PROGRAM

The following example program illustrates the use of the BINARY statements in an account ledger entry and display application.

```
BDEFS "DSC2:ACCOUNT/LDG"      !ENTERED IN KEYBOARD MODE

 10  DIM DAT(1),NAM(3)
 20  BOPEN "DSC2:ACCOUNT/LDG",LDG
 25  RESTOR #LDG
 30  PRINT @"C";"ACCOUNT LEDGER DATA ENTRY"
 40  LIM = 500                  !SET CREDIT LIMIT TO BE $ 500
 49  REM
 50  REM - INPUT VALUES AND PERFORM CALCULATIONS
 51  REM
 60  INPUT "INPUT DATE (MN/DY/YR) ",$DAT(0)
 70  INPUT "INPUT NAME ",$NAM(0)
 80  IF $NAM(0)="9999" THEN GOTO 120
 90  INPUT "PREVIOUS BALANCE = ";PRV;"  NEW AMOUNT = ";AMT
100  BAL=PRV+AMT                !BALANCE = PREVIOUS BAL + NEW AMOUNT
110  CRL=LIM-BAL                !CREDIT LIMIT = LIMIT - BALANCE
119  REM
120  REM - OUTPUT DATA, NAME, PRV, AMT, BAL, AND CRL TO BINARY FILE
121  REM
130  BINARY 1,LDG,12;2,DAT(0)::BINARY 1,LDG,24;2,NAM(0)
140  BINARY 2,PRV,AMT,BAL,CRL
150  IF $NAM(0)="9999" THEN GOTO 170
160  PRINT :: PRINT "MORE?(ENTER 9999 FOR NAME IF DONE) ":: GOTO 70
170  BCLOSE LDG                 !CLOSE FILE WHEN THROUGH WRITING
180  BOPEN "DSC2:ACCOUNT/LDG",LDG  !OPEN FILE TO BE READ
185  RESTOR #LDG
189  REM
190  REM OUTPUT HEADER ON PRINT DEVICE OR AT TOP OF SCREEN
191  REM
200  PRINT @"C"::PRINT
210  PRINT TAB(36);"PREVIOUS";TAB(46);"PURCHASE";
220  PRINT TAB(58);"  NEW "; TAB(69); "UNUSED"
230  PRINT "DATE";TAB(12);"NAME";TAB(36);"BALANCE";TAB(47);"AMOUNT";
240  PRINT TAB(58);"BALANCE";TAB(69);"CREDIT"
249  REM
250  REM - INPUT BINARY VALUES FROM FILE
251  REM
260  BINARY 1,LDG,12;3,DAT(0) :: BINARY 1,LDG,24;3,NAM(0)
270  BINARY 3,PRV,AMT,BAL,CRL
279  REM
280  REM - OUTPUT DATA VALUES
281  REM
290  IF $NAM(0)="9999" THEN GOTO 330
300  PRINT $DAT(0);TAB(12);$NAM(0);TAB(36);
310  PRINT #"$,$$$.99E";PRV;TAB(46);AMT;TAB(57);BAL;TAB(67);CRL
```

```
320  GOTO 250
330  BCLOSE LDG
340  STOP
```

This program requests the user to enter the date, name balance, and new purchase amount. It then stores these va the binary file "DSC2:ACCOUNT/LDG". It continues to ac inputs, until the user enters a "9999" in response to the "I prompt; the program then writes a terminator to the file the device/file. This is required before this file can be since the file was defined to be a sequential file. Next t opened, a header message is output, and the data values stor file are read and displayed until the end-of-file ("9999" i field) is encountered.

Execution of the previous program would produce the followin All user responses are underlined.

```
RUN
ACCOUNT LEDGER DATA ENTRY
INPUT DATE (MN/DY/YR) : 11/10/78
  INPUT NAME : JOHN DOE
PREVIOUS BALANCE = 100  NEW AMOUNT = 125

MORE? (ENTER 9999 FOR NAME IF DONE)
  INPUT NAME : JANE SMITH
PREVIOUS BALANCE = 400  NEW AMOUNT = 250

MORE? (ENTER 9999 FOR NAME IF DONE)
  INPUT NAME : 9999
```

DATE	NAME	PREVIOUS BALANCE	PURCHASE AMOUNT	NEW BALANCE	UNU CRE
11/10/78	JOHN DOE	$ 100.00	$ 125.00	$ 225.00	$ 27
11/10/78	JANE SMITH	$ 400.00	$ 250.00	$ 650.00	$ 15

```
STOP AT 340
```

SECTION VI

CHARACTER STRINGS

6.1 GENERAL

ASCII character strings are stored in the same variables as are other POWER BASIC variables. Variables are designated as containing character strings by program content or semantics. Any variable or array may contain ASCII characters and, in fact, may be filled with ASCII characters and numbers at the same time. String variables are designated by preceding the variable name with a dollar sign. Otherwise, the variable is treated as a number. ASCII characters are stored in contiguous memory locations with a null character terminating the string. You must ensure (with a DIM statement) that enough memory for a string variable has been set aside to store all the characters or other contiguous variables may be destroyed. The following formula indicates the number of ASCII characters you may store in any variable or array:

Configurable POWER BASIC

Number of characters = 6 x (number of variable elements) - 1

Examples:

```
I1        6 x 1 - 1 = 5
A(10)     6 x 11 - 1  = 65
N(10,5)   6 x (11 x 6) - 1 = 395
```

6.2 CHARACTER ASSIGNMENT

When a string assignment is made the actual characters are moved to the new variable.

Form:

```
$ VAR = <$VAR>
$ VAR =  "<character string>"
```

Characters are transferred one by one until a null byte is found.

Examples:

```
10  $I1="YES"
20  $J0=$J1
30  $N(4,0) = "CHARACTER STRING"
```

A character string is referred to as <$VAR> and implies either a literal string or a dollar sign preceding a variable. $<VAR> implies a

character string only of the form dollar sign preceding a variable

ASCII comparisons of the following form are valid:

IF <$VAR> RELATION <$VAR> THEN <BASIC STATEMENT>

Examples:

```
100  IF $I1="Y"  THEN GOTO 500
110  IF $N(I,0) =$N(J,0) THEN GOSUB 600
```

An ASCII variable may appear in a READ statement if the correspon
DATA statement entry is also an ASCII variable or an ASCII str:
When data types do not match you receive an error at the line nu
of the READ statement.

Example:

```
10  READ $N(0),A,B,$Z(0)
20  STOP
30  DATA "STRING DATA", 12345,A*10,$N(0)
```

In this example, $N(0) receives the character string "STRING DAT
the variable A receives the number 12345, and B the number 1234
Finally, the ASCII variable $Z(0) receives the same string as $N(0)

A dimensioned string variable can have a byte index into the charac
string by following the subscripts with a semicolon and the b:
displacement. The range of the index is from 1 through the last b
of the ASCII string. $A(0;1) is equivalent to $A(0).

Example:

```
10  DIM A(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  PRINT $A(0)
40  PRINT $A(0;1)
50  PRINT $A(0;10)
60  STOP

RUN
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
 JKLMNOPQRSTUVWXYZ

 STOP AT 60
```

Example:

```
10  DIM A(10),B(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  $B(0)=$A(0;10)
40  $A(0;2)=$B(0;2)
50  PRINT $A(0), $B(0)
60  STOP

RUN
 AKLMNOPQRSTUVWXYZ  JKLMNOPQRSTUVWXYZ

 STOP AT 60
```

6.3 CHARACTER CONCATENATION

Strings are concatenated by using the "+" operator.

Form:

$<VAR> = <$VAR> + <$VAR>+....

Concatenation operations may be chained together and the final string will automatically be terminated with a null by POWER BASIC.

Example:

```
10  DIM A(10)
20  $A(0)="ABCDE"
30  $A(0)=$A(0)+"FG"+"HIJK"
40  PRINT $A(0)
50  STOP

RUN
 ABCDEFGHIJK

 STOP AT 50
```

The following example results in a phenomenon called "CHOO-CHOO". It is caused because a null cannot be found.

```
10  $A(0)="ABCD"+$A(0)
```

POWER BASIC will detect this situation and terminate the string assignment by inserting a null when a previously stored value is again being selected for storage.

6.4 CHARACTER PICK

Characters can be picked from one variable into another by using the assignment operator.

Form:

```
$<VAR> = <$VAR> , <EXP>
```

The expression is evaluated and the resulting number specifies the number of bytes to be assigned. The string is then terminated with a null byte after the last character of the "picked" string.

Example:

```
10  DIM A(10),B(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  $B(0)=$A(0;4),6
40  $B(0;5)=$A(0),1
50  PRINT $B(0)
60  STOP

RUN

 DEFGA

 STOP AT 60
```

6.5 CHARACTER REPLACEMENT

Character replacement is very similiar to character pick with the exception that a null is not placed at the end of the string.

Form:

```
$<VAR>= <$VAR>;<EXP>
```

Example:

```
10  DIM A(10),B(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  $B(0)=$A(0;4),6          ! PICK 6 CHARACTERS
40  $B(0;5)=$A(0);1          ! REPLACE 1 CHARACTER
50  PRINT $B(0)
60  STOP

RUN
 DEFGAI

 STOP AT 60
```

6.6 CHARACTER INSERTION

Characters can be inserted into a string variable by using the slash (/) operator.

Form:

```
$ <VAR> =/ <$VAR>
```

The string is inserted without a null.

Example:

```
10  DIM A(10)
20  $A(0)="ABCDEFG"
30  $A(0;4)=/"..."
40  PRINT $A(0)
50  STOP

RUN
 ABC...DEFG

 STOP AT 50
```

6.7 CHARACTER DELETION

Characters are deleted from a string variable by using the same divide operator followed by an expression.

Form:

```
$<VAR> =  / <EXP>
```

The evaluated expression indicates the number of characters to be deleted.

Example:

```
10  DIM A(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  $A(0;5)= /10
40  PRINT $A(0)
50  STOP

RUN
 ABCDOPQRSTUVWXYZ

 STOP AT 50
```

6.8 BYTE REPLACEMENT

Individual bytes may be altered by using the numeric equivalent of an ASCII character along with the "%" operator.

Form:

$\$\langle VAR\rangle = \%\langle EXP\rangle \ldots$

Byte replacements may be chained together. The byte value may be specified as either a hexadecimal or decimal value. The evaluated expression specifies the byte code to be placed in the string variable. A null terminating byte is not placed at the end of the string. In some applications, the user should manually place the terminating null byte on the end of string, as in the following example.

Example:

```
10  DIM A(10)
20  $A(0)=%41%42%00
30  PRINT $A(0)
40  STOP

RUN
AB

STOP AT 40
```

6.9 CONVERT ASCII CHARACTER TO NUMBER

A character string may be converted to a number by using the assignment operator along with an error variable.

Form:

$\langle VAR\rangle = \langle \$VAR\rangle \ , \langle VAR\rangle$

The delimiting character is placed in the first byte of the error variable. Hence, the conversion routine was successful in converting the whole string if a null was the resulting delimiter.

Example:

```
10  $NUM="12DE"
20  N="1234",E
30  N1=$NUM,E1
40  PRINT N,$E
50  PRINT N1,$E1
60  STOP
```

```
RUN
 1234
 12                D
STOP AT 50
```

Note that the first numeric character string was successfully converted with no invalid characters encountered. However, the second numeric character string converted only the string "12", and placed the non-numeric "D" character into the variable E1.

6.10 CONVERT NUMBER TO ASCII CHARACTER

A number can be converted to a string simply by assigning the number to a string variable.

Form:

```
$ <var> = <exp>
```

The string will properly be terminated with a null.

Example:

```
10  DIM A(10),B(10)
20  $A(0)=4*ATN(1)
30  $B(0)= SQR(2)
40  PRINT $A(0), $B(0)
50  STOP

RUN
 3.141592       1.414213

 STOP AT 50
```

Formatted conversions can also be made by preceding the expression with the formatting operator "#" and a string. The form is:

```
$<VAR> = # <$VAR> , <EXP>
```

The formatting rules are the same as those given under print formatting. (See paragraph 5.8.2.1.)

Example:

```
10  DIM A(10),B(10)
20  $A(0)=#"999,990.99",1234
30  $B(0)=#"<<<,<<<.00>",-1234
40  PRINT $A(0),$B(0)
50  STOP

RUN
   1,234.00      <1,234.00>

 STOP AT 50
```

6.11 STRING LENGTH FUNCTION

The length of a string variable is returned by using the LEN function.

Form:

```
LEN(<$VAR>)
```

A zero is returned if the string is the null string.

Example:

```
10  DIM A(10),B(10)
20  $A(0)=""
30  $B(0)="ABCDEFGUIJKLMNOPQRSTUMVWYZ"
40  PRINT LEN($A(0)),LEN($B(0))
50  STOP

RUN
 0           26

STOP AT 50
```

6.12 CHARACTER SEARCH FUNCTION

To search for a given string, use the SRH function.

Form:

```
SRH (<$VAR>,<$VAR>)
```

The function returns the character position indicating where the first string is located in the second string. If the search is unsuccessful, a zero is returned.

Example:

```
10  DIM A(10),B(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  $B(0)="ZYXWVUTSRQPONMLKJIHGFEDCBA"
40  S1=SRH("EFG",$A(0))
50  S2=SRH("EFG",$B(0))
60  PRINT S1,S2
70  STOP

RUN
 5           0

STOP AT 70
20  $SET(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
STOP AT 70
```

Example:

```
10  DIM SET(5)
    20  $SET(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"30  PRINT
    30  PRINT
    40  POS=SRH($CHR,$SET(0))
    50  POS=SRH($CHR,$SET(0))
    60  IF POS=0 THEN GOTO 30
    70   ELSE $CHA=$SET (0;POS),1
    80  PRINT "POSITION = ";POS,"CHARACTER= ";$CHA
    90  GOTO 30
   100  STOP

    RUN
    INPUT CHARACTER Z
    POSITION=26 CHARACTER=Z
    INPUT CHARACTER 4
    INPUT CHARACTER
```

6.13 CHARACTER MATCH FUNCTION

When looking for character agreement, the MCH function can be used to return the number of characters which are the same for two strings. Form:

```
MCH (<$VAR>, <$VAR>)
```

A zero is returned if a match is not found.

Example:

```
10  DIM A(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  PRINT MCH("ABCDXYZ",$A(0)),MCH("BCGH",$A(0;2))
40  STOP


RUN
 4          2

STOP AT 40
```

6.14 ASCII CHARACTER CONVERSION FUNCTION

The ASC function returns the ASCII decimal numeric value of the first character of the specified string variable.

Form:

ASC($<VAR>)

The ASC function is the inverse of the byte replacement operator (%), i.e., $A = %ASC($A).

The following example takes the upper case string in the variable $A(0) and converts it to the lower case string in the variable $B(0) using the ASC function to obtain the decimal ASCII code for the character conversions.

Note that the VDT913 video display terminal does not support lower case characters, therefore this example will not function correctly when executed on the Host POWER BASIC using the VDT913 as the terminal device.

Example:

```
10  DIM A(10),B(10)
20  INPUT "INPUT STRING", $A(0)
30  FOR I=1 TO LEN ($A(0))
40   $C=$A(0;I),1
50  IF $C=" " THEN $B(0;I)=" "
60   ELSE $B(0;I)=%(ASC($C)+020H)%0
70   NEXT I
80   PRINT $A(0)
90   PRINT $B(0)
100  STOP
```

```
RUN
    INPUT STRING:  UPPER CASE TO LOWER CASE
    UPPER CASE TO LOWER CASE
    upper case to lower case
    INPUT STRING:
```

SECTION VII

POWER BASIC FUNCTIONS

7.1 GENERAL

POWER BASIC includes several predefined mathematical, string, and miscellaneous functions. The Host POWER BASIC Interpreter supports all the functions presented below, while the Target or "Configured" POWER BASIC Interpreter supports all of the functions with the exception of FRA, SGN, and TAN. These functions may be used in an application on the Host POWER BASIC Interpreter; however, they may not be configured into a Target POWER BASIC Interpreter and application.

A function is called by using the following form in any statement where a variable may be used:

function name (<argument>)

where

function name is a three-letter name
argument may be an expression or variable.

The specified function of the argument replaces the function name in the statement in which it is used. Functions may be used instead of, or in combination with, variables in almost all POWER BASIC statements such as: assignment, PRINT, IF, FOR, ON, DEF, etc.

7.2 MATHEMATICAL FUNCTIONS

Paragraphs (7.2.1 through 7.2.10) describe the mathematical functions and their associated forms provided by POWER BASIC.

7.2.1 ABSOLUTE VALUE FUNCTION (ABS)

The absolute value function (ABS) obtains the absolute value of a positive or negative number. The argument entered following the function name is the variable name or numeric value for which the absolute value is required. The function returns a non-negative argument unaltered and returns the absolute value of a negative argument.

Example:

```
10  INPUT X
20  PRINT SQR(ABS(X))
30  STOP
```

7.2.2 ARCTANGENT FUNCTION (ATN)

The argument entered following the function name is the ratio representing a tangent function. The function returns the corresponding angle in radians. Multiply the number of radians by 180/3.14159265 (Pi) to obtain the angle in degrees.

Example:

```
10  INPUT X
20  D = ATN(X)*(180/3.14159265)
30  PRINT D
40  STOP
```

Executing the above example produces:

```
? 5.9246
 80.419473
```

7.2.3 SINE AND COSINE FUNCTIONS (SIN)(COS)

The argument entered following the function name represents an angle in radians. When the angle is measured in degrees, multiply the number of degrees by 3.14159265 (Pi) /180 to obtain the angle in radians. The function determines the quadrant corresponding to the argument and returns the function value.

Example:

```
10  INPUT N
20  PRINT SIN(N);COS(N);
30  STOP
```

Executing the above example produces:

```
? 1.25
 0.94898462  0.31532236
```

7.2.4 EXPONENTIAL FUNCTION (EXP)

The argument entered following the function name is an exponent of e (the base of natural logarithms). The function returns the value of e raised to the power specified in the argument.

Example:

```
10  INPUT E
20  PRINT EXP(E)
30  STOP
```

Executing the above example produces:

```
? 25
 7200489900
```

7.2.5 FRACTIONAL PART FUNCTION (FRA)

The fractional part function (FRA) returns the signed fractional portion of the argument. Effectively, FRA(X)=X-(INP(X)).

Example;

```
10  INPUT E
20  PRINT FRA(E)
30  STOP
```

Executing the above example produces:

```
? 3.1415926
 .14159260
```

Target POWER BASIC Interpreter: The FRA function is not supported by the POWER BASIC Configurator. Therefore, the FRA function should not be present in a final application which is to be configured into a customized (Target) POWER BASIC Interpreter.

7.2.6 INTEGER PART FUNCTION (INP)

The integer part function (INP) returns the signed integer portion of the argument. The INP function is useful in modular arithmetic and for correcting errors resulting from truncation or rounding of functions. The argument entered following the function name is the value for which the integer portion is required.

```
10  A=3 2
20  PRINT "A = ";A, "INP(A)= ";INP(A),
30  PRINT "INP(A+1E-07)= ";INP(A+1E-07)
40  STOP
```

Executing the previous example procedues:

```
A = 9     INP(A) = 8      INP(A+1E-07) = 9

STOP AT 40
```

7.2.7 LOGARITHM FUNCTION (LOG)

The argument entered following the function name is the value for which the natural logarithm (base e) is required. The function returns the natural logarithm of the argument. Attempts to find the logarithm of a non-positive argument will result in an error. (LOG OF NON-POSITIVE NUMBER).

Example:

```
10  INPUT L
20  PRINT LOG(L)
30  STOP
```

Executing the above example produces:

```
? 5280
 8.5716814
```

7.2.8 SIGN FUNCTION (SGN)

The sign function (SGN) tests the sign of a value. The argument entered following the function name is the value to be tested. The function returns +1 when the argument is a positive number, or -1 when the argument is a negative number. The function returns zero when the argument is zero.

Example:

```
10  INPUT A
20  IF SGN(A) 0 THEN GOTO 50
30  PRINT LOG(A)
40  GOTO 10
50  PRINT "LOG VALUE UNAVAILABLE"
60  STOP
```

Executing the above examples produces:

```
? 10
 2.3025851
?-20
 LOG VALUE UNAVAILABLE

STOP AT 60
```

Target POWER BASIC Interpreter: The SGN function is not supported by the POWER BASIC Configurator. Therefore the SGN function should not be present in a final application program which is to be configured into a customized (Target) POWER BASIC Interpreter.

7.2.9 SQUARE ROOT FUNCTION (SQR)

The square root function (SQR) returns the square root value of the specified argument. The argument entered following the function name may be positive or zero. The function returns the square root of the argument. An error message (SQUARE ROOT OF NEGATIVE NUMBER) is produced if the argument is negative.

Example:

```
10  INPUT K
20  PRINT SQR(K)
30  STOP
```

Executing the above example produces:

```
? 2
 1.4142136
```

7.2.10 TANGENT FUNCTION (TAN)

The argument entered following the tangent function (TAN) represents an angle in radians. When the angle is measured in degrees, multiply the number of degrees by 3.14159265(Pi)/180 to obtain the angle in radians. The function determines the quadrant corresponding to the argument and returns the tangent.

Example:

```
10  INPUT M
20  PRINT TAN(M)
30  STOP
```

Executing the above example produces:

```
? 0.137
 0.13786360
```

Target POWER BASIC Interpreter: The TAN function is not supported by the POWER BASIC Configurator. Therefore the TAN function should not be present in a final application which is to be configured into a customized (Target) POWER BASIC Interpreter.

7.3 STRING FUNCTIONS

The string functions described in Paragraphs 7.3.1 through 7.3.4 may be employed in POWER BASIC programming.

7.3.1 ASCII CHARACTER CONVERSION FUNCTION

The ASCII character conversion function (ASC) returns the decimal ASCII numeric value of the first character of the specified string. For additional details, refer to Section 6, Paragraph 6.14.

Example:

```
10    $A="B"
20    B=ASC[$A]
30    $C=%B+020H
40    D=ASC[$C]
50    PRINT $A,B,$C,D
60    STOP

RUN
B                66               b               98

STOP AT 60
```

7.3.2 STRING LENGTH FUNCTION (LEN)

The string length function (LEN) returns the number of non-null characters starting at the evaluated variable address. The argument of the LEN function must be specified as a string by either the $ or "string constant" operators. For additional details, refer to Section 6, paragraph 6.11.

Example:

```
10  $I="ABC"
20  J=LEN($I)
30  K=LEN("ABCDEFGHIJKLMNOP")
40  PRINT J,K
50  STOP
```

Executing the above example produces:

```
3    16
```

7.3.3 CHARACTER MATCH FUNCTION (MCH)

The character match function (MCH) returns the number of characters to which the two strings agree. A value of zero indicates no match. For additional details, refer to Section 6, paragraph 6.13.

Example:

```
10  $C="ABCD"
20  M=MCH("AB",$C)
30  PRINT M
40  STOP
```

Executing the above example produces:

```
2          (RESULT)
```

7.3.4 CHARACTER SEARCH FUNCTION (SRH)

The character search function (SRH) returns the character position of string 1 in string 2. A character position of zero indicates an unsuccessful search. For additional details, refer to Section 6, Paragraph 6.12.

Example:

```
10  $C = "ABCD"
20  S= SRH ("BC",$C)
30  PRINT S
40  STOP
```

Executing the above example provides:

```
2      (RESULT)
```

7.4 MISCELLANEOUS FUNCTIONS

The miscellaneous functions described in paragraphs 7.4.1 through 7.4.8 are supported by POWER BASIC.

7.4.1 CRU SINGLE BIT FUNCTION (CRB)

A CRU bit, addressed relative to a base displacement, is either read or stored according to program context. The displacement ranges from -128 to +127. (Refer to Section 5, paragraph 5.10 for details on the BASE statement.) The function returns a 1 if the CRU bit is set, and a 0 if not set. Likewise, the selected CRU bit is set to 1 if the assigned value is non-zero and to 0 if the assigned value is zero. For example:

```
CRB(10)=0
```

will clear the tenth bit relative to the base, while

```
CRB(11)=1 or CRB(11)=345
```

will set the eleventh bit on. Also,

```
IF CRB(5) THEN J=4
```

will set J=4 if the fifth bit is 1.

7.4.2 CRU FIELD FUNCTION (CRF)

The specified number of bits are transferred to or read from the CRU starting at the address set by the BASE statement. (Refer to Section 5, paragraph 5.10 for details on the BASE statement.) The specified number of bits ranges from 0 to 15. If zero, all 16 bits will be transferred. For example:

```
CRF(0) = -1
```

transfers 16 bits (hex 'FFF') to the CRU address specified by the BASE statement. While,

```
VAL=CRF(8)
```

reads 8 bits from the CRU base address and stores the result in VAL.

7.4.3 KEY FUNCTION (NKY)

The key function (NKY) conditionally samples the keyboard in run time mode. When the argument is zero the decimal value of the last key struck is returned and the key register is reset. A value of zero is returned if none of the keys were struck. If the argument is non-zero, the argument is compared with the last key struck. If they are the same, a value of 1 is returned and the key register is reset. Otherwise, a value of 0 is returned. For example,

```
I = NKY(0)
```

returns the last key struck, or a 0 if none of the keys were struck; while

```
IF NKY(041H) THEN PRINT "A"
```

prints "A" if the last key entered was "A". The argument value may be expressed as either a decimal or hexadecimal numeric constant or variable.

7.4.4 SYSTEM INTERROGATION (SYS) FUNTION

The system interrogation function (SYS) obtains system parameters generated during program execution. For example,

```
A = SYS(0)
```

returns the control character entered during either numeric or string variable assignment when using the question mark (?) operator of the INPUT statement. (Refer to the INPUT statement, Section 5, paragraph 5.8.1.2.)

```
A = SYS(1)
```

returns the ERROR code number when an error is encountered and is used with the ERROR statement of Section 5, paragraph 5.6.6.

```
A = SYS(2)
```

returns the statement number in which the error occurred and is used with the ERROR statement of Section 5, paragraph 5.6.6.

7.4.5 DELTA TIME (TIC) FUNCTION

The delta time (TIC) function samples a real time clock and returns the current TIC value minus the expression value. For example:

```
T = TIC(0)
```

obtains current time, and

```
D = TIC(T)
```

calculates elapsed time since the time stored in the variable T (i.e., TIC (T) = TIC (0) - T.

The TIC function is one of the system dependent POWER BASIC features which operates differently on the Host POWER BASIC Interpreter on the FS990 system than on the Target POWER BASIC Interpreter on the TM990 board based system. This implies that an application program using the TIC function will operate differently when executed on the FS990 system than in the "Configured" POWER BASIC Interpreter and application which is to reside on the TM990 board system. The user must be aware of the differences when developing a program on the Host POWER BASIC Interpreter and appropriately compensate for them before executing the Configuration process. That is, the user must modify the use of the TIC function in the application program to work correctly with the Target POWER BASIC Interpreter before performing the

Configuration process. The differences between the systems are presented below.

Host POWER BASIC Interpreter:

The TIC function of the Host POWER BASIC Interpreter utilizes the Date and Time Supervisor Call of the TX990 Operating System to obtain the current time of day and then calculates the current time value minus the expression value. The time value returned by the Supervisor Call has a maximum resolution of one second. Also note that the FS990 system clock is always running (ie., it does not require the "TIME 0" statement to start the clock).

The following example program will output a bell (ASCII code "07") on the terminal device once every five-seconds.

```
LIST
 10  REM  THE CLOCK DOES NOT NEED TO
 11  REM  BE STARTED USING THE HOST POWER BASIC INTERPRETER
 20  $B=%07%00  !ASCII CODE FOR BELL
 30  A=TIC(0)
 40  IF TIC(A)<5 THEN GOTO 40
 50  PRINT $B;
 60  GOTO 30
```

Target POWER BASIC Interpreter:

The TIC function used in the customized (Target) POWER BASIC Interpreter utilizes the real time clock of the TMS9901 Programmable Systems Interface on the TM990/100M or TM990/101M microcomputer board. The TMS9901 is programmed to generate an interrupt (or TIC) every 40 milliseconds (1/25th of a second) with a system clock rate of 3 MHz. This results in a maximum resolution of 40 milliseconds in the final application. Note that TMS9901 clock must be started by the user through execution of the "TIME 0" statement or by setting the clock via the "TIME <exp>,<exp>,<exp>" statement for the TIC function to return meaningful values.

The following example program when "Configured" in a POWER BASIC application will output a bell (ASCII code "07") on the terminal device once every five seconds.

```
LIST
 10  TIME 0      ! THIS WILL START THE CLOCK
 20  $B=%07%00   ! ASCII CODE FOR BELL
 30  A=TIC(0)
 40  IF TIC(A) <5*25 THEN GOTO 40
 50  PRINT $B;
 60  GOTO 30
```

7.4.6 MEMORY INTERROGATE/MODIFY (MEM) FUNCTION

The memory interrogate/modify (MEM) function reads or modifies a memory location (byte) as specified by the argument. This argument may be expressed as either a decimal or hexadecimal numeric constant or variable. For example:

```
M = MEM(0FF00H)
```

reads the byte from location hex "FF00", while

```
MEM(0FF00H) = 15
```

stores a decimal 15 (hex "F") at location hex "FF00".

The following example POWER BASIC program will dump a specified area of memory in 16-byte fields, with the ASCII byte equivalents displayed to the right.

```
LIST
 100  PRINT @"C";:: PRINT :: PRINT "          MEMORY DUMP PROGRAM"
 110  PRINT
 120  INPUT "MEMORY START ADDRESS = ";ST
 130  INPUT "  MEMORY END ADDRESS = ";END
 140  PRINT
 150  PRINT "DUMP OF MEMORY FROM ";#,ST;" TO ";#,END
 160  PRINT
 165  ST=2*INP(ST/2)
 170  FOR I=ST TO END STEP 16
 180   PRINT #,I;" = ";
 190   FOR J=I TO I+14 STEP 2
 200    PRINT #;MEM(J);#;MEM(J+1);" ";
 210   NEXT J
 220   FOR J=I TO I+14 STEP 2
 225    PRINT "  ";
 230    IF MEM(J)<020H LOR MEM(J)>07EH THEN $CHR=".":: GOTO 250
 240    $CHR=%MEM(J)
 250    PRINT $CHR;
 260    IF MEM(J+1)<020H LOR MEM(J+1)>07EH THEN $CHR=".":: GOTO 280
 270    $CHR=%MEM(J+1)
 280    PRINT $CHR;
 290   NEXT J
 300   PRINT
 310  NEXT I
 320  PRINT :: PRINT
 330  STOP
```

```
RUN

                MEMORY DUMP PROGRAM

MEMORY START ADDRESS = 02000H
  MEMORY END ADDRESS = 02060H

DUMP OF MEMORY FROM 2000 TO 2060

2000 = 0202 44DC 06A0 2032 C020 44BC C080 0222   ..  D.  ..   2  .  D.  .. ."
2010 = 002A 06A0 2032 C010 16F9 9828 0009 4518   .*  ..   2  ..  ..  .(  ..  E.
2020 = 1A04 C00A 06A0 4958 C280 10CB 0200 7000   ..  ..  ..  IX  ..  ..  ..  P.
2030 = 108B 0201 0900 2FC1 C0C2 05C3 C103 C0D4   ..  ..  ..  /.  ..  ..  ..  ..
2040 = 1309 80C8 16FB C513 16F9 C484 C0E2 0002   ..  ..  ..  ..  ..  ..  ..  ..
2050 = 16F5 04D2 045B 069F 1005 10AB C050 C800   ..  ..  .(  ..  ..  ..  .P  ..
2060 = 48EC 10AF 0200 48FC 10F9 4E4F 5420 5255   H.  ..  ..  H.  ..  NO  T   RU


STOP AT 330
```

7.4.7 BIT MODIFICATION (BIT) FUNCTION

The bit modification (BIT) function reads or modifies any bit within a variable. The function returns a 1 if the bit is set and a 0 if not set. Likewise, the selected bit is set to 1 if the assigned value is non-zero, and to zero if the assigned value is zero. For example:

```
IF BIT(A,1) THEN PRINT "ON"
```

prints "ON" if bit 1 of variable A is on; while

```
BIT (A,2)=1 or BIT (A,2)=750
```

turns "on" the second bit of variable A.

Refer to Section 3, paragraph 3.7.7 for an application of the BIT function.

7.4.8 RANDOM NUMBER (RND) FUNCTION

The random number function (RND) is used to generate a psuedo random number between 0 and 1. For example:

PRINT RND

would return a random number like

.2113190

Refer to the RANDOM statement paragraph 5.12 for additional information.

SECTION VIII

POWER BASIC CONFIGURATION PROCESS

8.1 INTRODUCTION

The user will utilize the enhanced statements, commands, and functions of the Host POWER BASIC Interpreter when developing application programs which are designed for execution in a final or "Target" TM990 board environment. Once the user is reasonably confident that the program developed on the Host POWER BASIC Interpreter will perform the intended TM990 application, the user will SAVE the program in a diskette file, and then perform the Configuration process. This process will produce an "application program customized" POWER BASIC run-time module. The configuration process consists of first executing the POWER BASIC Configurator, followed by execution of the TX990 Link Editor. This process results in a faster and much smaller implementation of the POWER BASIC Interpreter linked with the original application program. The resultant run-time module is appropriately ROM/RAM partitioned for execution in a TM990/100M or TM990/101M microcomputer board system, and is appropriately termed the Target POWER BASIC application.

8.2 TYPICAL CONFIGURATION CYCLE

The POWER BASIC user should develop application programs using the techniques, statements, commands, and functions as described in Sections 3 through 7 of this manual. When final development and testing are as complete as possible without actual interface to the final target application, the user will perform the configuration process as outlined below. The Configurator reads the specified POWER BASIC application program, produces the Link Control file, root module, and listing outputs. The TX990 Link Editor then uses this Link Control File and root module to produce the final linked object module (Target POWER BASIC application). The Linked Object module will then typically be tested in the actual target application through use of the AMPL system. If the program functions correctly, this object module will then be programmed into EPROM's and mounted in the target TM990 system. The overall configuration cycle is shown in Figure 8.1 on the following page.

8.3 POWER BASIC CONFIGURATOR

The POWER BASIC Configurator enables the user to generate an application customized POWER BASIC run-time module. This run-time module contains both the customized POWER BASIC Interpreter and the user's BASIC application program and is appropriately ROM/RAM partitioned for execution in a TM990 system. The Interpreter is customized at the BASIC Statement and Function level with the

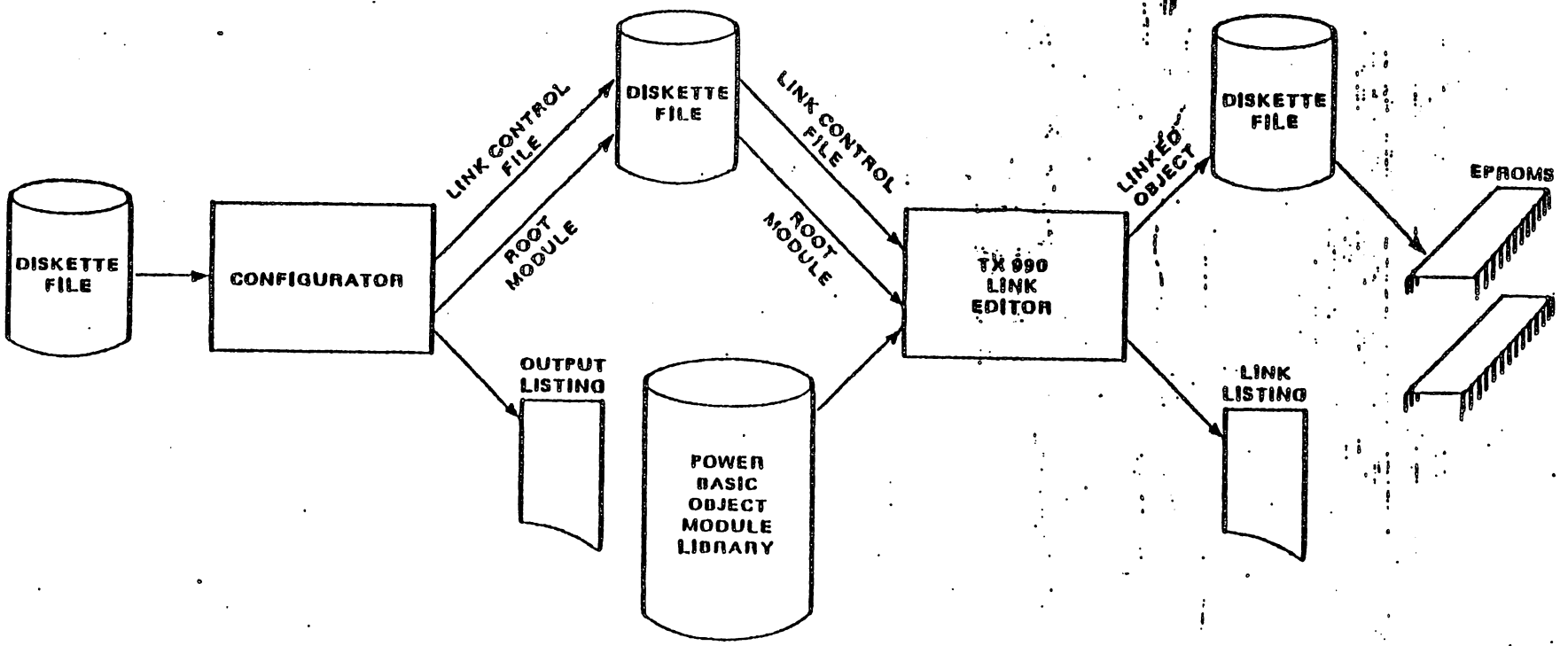


FIGURE 8-1. APPLICATION PROGRAM CONFIGURATION PROCESS

specified POWER BASIC application program, and results in a minimum TM990 system memory (ROM) requirement for the specified application. The Configurator scans the POWER BASIC application, and "remembers", via the Link Control File, each of the POWER BASIC statements and functions used in the program. It then produces: 1) the corresponding Link Control File, 2) the ROOT module containing the pseudocode representation of the POWER BASIC application program, and 3) a listing for a hard copy of the configurator execution. The Link Control File will "INCLUDE" only those modules required to execute the application program. The Target (Configured) POWER BASIC Interpreter does not contain the "editor" (which performs statement entry or modification), the "lister" (which performs listing of a BASIC program), any of the POWER BASIC commands, or any unused statements of the POWER BASIC language.

8.3.1 POWER BASIC APPLICATION PROGRAM

The user will develop application programs on the Host POWER BASIC Interpreter with the final goal of "configuration" into a customized application using the POWER BASIC Configurator. Several of the POWER BASIC statements and functions of the Host POWER BASIC Interpreter are not supported by the Configurator and therefore cannot be included in a final application program intended for configuration. In addition, several statements and functions operate differently on the Host POWER BASIC Interpreter than in a Target POWER BASIC Interpreter/Application. These will be explained in the paragraphs below.

8.3.1.1 NON-CONFIGURABLE STATEMENTS AND FUNCTIONS

The Interpreter of the Target POWER BASIC application does not contain the editor function of POWER BASIC, therefore a POWER BASIC program may not be entered into a "configured" interpreter. This greatly reduces the amount of overhead required in the final application, and enables a much smaller interpreter/application module to be developed for execution in TM990 applications where the program will not be modified and an editor is not required. Since an editor is not present, two of the statements of the POWER BASIC language are no longer required, and these are not supported by the Configurator and may not be configured into a Target application. The following statements fall into this category:

ESCAPE
NOESC

Similarly, none of the POWER BASIC commands may be configured into a target application, since commands cannot be entered into a program and they are only used during program development. Therefore the following commands of the Host POWER BASIC Interpreter cannot be

included in any target applications:

CONTINUE	RUN
LIST	SAVE
LOAD	SOURCE
NEW	STACK
NUMBER	SIZE
PURGE	

Also, none of the file management statements of the Host POWER BASIC Interpreter may be entered into an application program since there is no file support in the final target application. The following statements fall into this category:

BDEFS	RESET
BDEFR	RESTOR # <var>
BDEL	BINARY 1
BOPEN	BINARY 2
BCLOSE	BINARY 3
COPY	BINARY 4

In addition, the following statements and functions of the Host POWER BASIC Interpreter cannot be configured into a target application. These are not supported by the Configurator so that the resultant "ROOT module" can remain transportable to Development POWER BASIC (part no. TM990/451). That is, the user can take the pseudo code representation of the application program contained in the "ROOT module", program it into EPROM, and place it in the Development POWER BASIC system. It may then be accessed and executed via the "LOAD address" command of Development BASIC. For additional information refer to paragraph 8.4.3.1 below or to the POWER BASIC REFERENCE MANUAL, MP308. To maintain this compatiblity, the following statements and functions cannot be configured into a Target POWER BASIC application.

BYE	FRA
DIGITS	SGN
EQUATE	TAN
SPOOL	<cursor positioning>

If any of the statements or functions presented in this section are encountered during Configurator execution, an error code will be generated and the offending statement will be displayed. This statement will not be configured into the application, and the Configurator will continue with the next statement of the application program.

8.3.1.2 SPECIAL STATEMENTS AND FUNCTIONS

Some of the system dependent POWER BASIC statements and functions

operate differently on the Host POWER BASIC Interpreter on the FS990 than on the Target POWER BASIC Interpreter in a TM990 board based system. An application program using these statements or functions will function in a similar manner. However, it will operate in a slightly different manner when executed on the Host Interpreter than when on the "Configured" POWER BASIC Interpreter and application (which is to reside on the TM990 board system). The user must be aware of the differences when developing a program on the Host POWER BASIC Interpreter and appropriately compensate for their use before executing the Configuration process. That is, the user must modify the use of these statements and functions in the application program to work correctly with the Target POWER BASIC application before performing the Configuration process.

The statements and functions which fall into this category are presented below. The user should reference the corresponding sections of this manual for the differences in their operation between systems.

BAUD	TIC
CALL	TIME
IMASK	TRAP
IRTN	UNIT
REM	<tail remarks>

Also note that the user MUST include the BAUD statement in the application program if any terminal input/output is required, and must also use the UNIT Statement to appropriately direct the output of the application to either or both ports A and B of the TM990/101M board.

8.4 CONFIGURATOR EXECUTION

The user will call the POWER BASIC Configurator into execution by performing the steps as outlined in Section 2, "LOADING AND EXECUTION OF HOST POWER BASIC INTERPRETER AND CONFIGURATOR". When the Configurator is first called into execution, the following banner message and user input prompt are displayed on the terminal device.

```
POWER BASIC CONFIGURATOR REV C.n.m

APPLICATION SRCE=
```

where,

n = the release number
m = revision number

A description of the prompts and associated user responses and entries is provided in the following paragraphs.

8.4.1 RESPONDING TO CONFIGURATOR PROMPTS

The user's response to the APPLICATION SRCE=, OBJECT FILE=, LINK CONTROL=, and LIST FILE= will specify the application program to be configured as well as the devices/files to which the subsequent outputs will be directed. When responding to the Configurator prompts, the user must use the TX990 pathname syntax to identify both devices and files. The TX990 pathname syntax is repeated below for user convenience.

8.4.1.1 PATHNAME SYNTAX

All TX990 pathnames adhere to the following rules:

- All device pathnames consist of a one to four character device name assigned to that device during system generation. Typical device pathnames are DSC, DSC2, LP, CS1, and CS2. The user must reference his system generation for the device names used in his particular TX990 Operating System.

- All diskette file pathnames consist of one to seven characters separated from the device name by a colon (:). The first character must be alphabetic (A-Z); the rest may be alphanumeric. The file name is followed by an extension to the file name, which is one to three characters separated from the file name by a slash(/). The first character must be alphabetic, the rest must be alphanumeric. The file name and extensions are specified when the file is created. Typical valid diskette file names are: DSC2:LOAD/ONE, DSC:PROGRAM/FIN, and :PROCESS/CNT. Invalid diskette file names would be DSC:TEMPERATURE/OBJ -- too many characters, DSC2:FILE/915 -- first character of extension must be alphabetic, and DSC:MOTOR/STEP -- extension too long.

The Configurator/TX990 interface checks the syntax of all file and device names as they are entered by the user. If the pathname syntax is illegal, an invalid pathname error code will result, and the prompt will be issued again. Similarly, if the specified device/file does not exist when attempted to be opened, an undefined file name error will result and the prompt will be issued again.

If a diskette file is specified to receive one of the outputs of the Configurator, it must have been previously defined by either the BDEFS POWER BASIC statement or by the create file (CF) command of the SYSUTL package.

For example:

```
BDEFS  "DSC2:PROCESS/OBJ"
BDEFS  "PROCESS/LNK"
```

```
BDEFS  "DSC2:PROCESS/LST"
```

or, similarly,

```
OP: CF,DSC2:PROCESS/OBJ
OP: CF,DSC2:PROCESS/LNK
OP: CF,DSC2:PROCESS/LST
```

8.4.1.2 APPLICATION SRCE= Prompt

The user's response to the APPLICATION SRCE= prompt will specify the pathname of the POWER BASIC application program to be configured. The Configurator will immediately open and rewind the specified device/file.

The user MUST enter a valid pathname of a device/file which contains a POWER BASIC application program before he may continue to the next prompt.

Typically, the application source will reside on the application diskette in the right-hand diskette drive (DSC2). The left-hand diskette drive (DSC) is reserved for the master or production diskette containing the Power Basic Configurator and Link Editor programs. Typical application source responses would be:

```
APPLICATION SRCE=DSC2:PROCESS/SRC
APPLICATION SRCE=DSC2:FILE1/PRG
APPLICATION SRCE=DSC2:MOTOR/SRC
```

8.4.1.3 OBJECT FILE= Prompt

The user's response to the OBJECT FILE= prompt will specify the pathname of the device/file to receive the "ROOT module" produced by the Configurator. This ROOT module contains the pseudo-code representation of the application program. The application program is translated into internal pseudocode by the "editor" (or translator) function of the Configurator. This pseudocode or ROOT module is then stored into the specified device/file for use by the TX990 Link Editor when producing the final Target POWER BASIC Interpreter/Application.

The user need not enter a response to the OBJECT FILE= prompt if this output is not required. However, in most cases the user will enter a valid pathname in response to the OBJECT FILE= prompt since this file is required to generate a final Target POWER BASIC application. If a diskette file is specified, it must have been previously defined by either the BDEFS POWER BASIC statement or by the create file (CF) command of the SYSUTL package.

Note that the left hand diskette drive is reserved for the master or production diskette, and the right-hand drive is used as the application diskette. Therefore, the user typically directs the OBJECT FILE= output (ROOT module) to the right-hand diskette (DSC2).

Typical Object File responses would be:

```
OBJECT FILE=DSC2:PROCESS/OBJ
OBJECT FILE=DSC2:FILE1/OBJ
OBJECT FILE=DSC2:MOTOR/OBJ
```

8.4.1.4 LINK CONTROL= Prompt

The user's response to the LINK CONTROL= prompt will specify the pathname of the device/file to receive the link control sequence produced by the configuration process. This control file will later be specified in response to the INPUT: prompt when the TX990 Link Editor is executed. The link control file will contain the appropriate LIBRARY, END, TASK, and INCLUDE commands to perform the final link of the Target POWER BASIC Interpreter/Application. It will contain INCLUDE commands for all object modules required by the statements and functions of the application program, as well as INCLUDE commands for the START module (core module of the Interpreter), the ROOT module (ie., the user response to the OBJECT FILE= Prompt), and any assembly language modules referenced by a CALL statement in the application program.

The user need not enter a response to the LINK CONTROL= prompt if this output is not desired. However, in most cases the user will enter a valid pathname for the LINK CONTROL=prompt since this output is required to generate the final Target POWER BASIC application. If a diskette file is specified, it must have been previously defined by either the BDEFS POWER BASIC statement, or by the create file (CF) command of the SYSUTL package.

Note that the left-hand diskette drive is reserved for the master or production diskette, and the right-hand drive is used as the application diskette. Therefore, the user typically directs the LINK CONTROL= output to the right-hand diskette (DSC2). Typical Link Control responses would be:

```
LINK CONTROL=DSC2:PROCESS/LNK
LINK CONTROL=DSC2:FILE1/LNK
LINK CONTROL=DSC2:MOTOR/LNK
```

8.4.1.5 LIST FILE= Prompt

The user's response to the LIST FILE= prompt will specify the pathname of the device/file to receive the listing output produced by the Configurator. The Configurator will immediately open and rewind the specified device/file. This list file will contain a complete listing of the configuration process. It will contain any "edit" errors which occurred, a list of statements and functions used by the application program, and a list of object modules "INCLUDED" by the Configurator.

The user need not enter a response to the LIST FILE prompt if this output is not desired. However, the user will typically enter a valid pathname for this prompt to receive a listing of the configuration

process. If a diskette file is specified, it must have been previously defined by either the BDEFS POWER BASIC statement or by the create file command of the SYSUTL package.

Typically the user will direct the LIST FILE= output directly to the line printer (LP) or to a diskette file on the application disk (DSC2). If directed to a diskette file, the listing may later be output on a printer device through use of the TXCCAT or LIST 80/80 utility program.

8.4.1.6 SPECIAL KEYBOARD CONTROL KEYS

The special keyboard control keys for the VDT911 and VDT913 terminals are described in Table 8-1. Note that the phrase "(ctrl)" indicates that the user holds down the control key while depressing the key corresponding to the character immediately following. (Refer to Figure 8-1.)

TABLE 8-1

SPECIAL CHARACTERS

VDT911	VDT913	FUNCTIONS
(ctrl) D (ctrl) 2 unlabeled function key *	RESET	Terminate Configurator execution and return to TX990 Operating System
RETURN ↓ (down arrow) ENTER (ctrl) J (ctrl) M	NEW LINE ↓(down arrow)	Enter user response
↑ (up arrow) (ctrl) X	↑(up arrow)	Backup to previous prompt
ERASE FIELD ERASE INPUT (ctrl) C	CLEAR DEL LINE BACK TAB	Clear current user response
DEL CHAR ←(left arrow) (ctrl) H	DEL CHAR ←(left arrow)	Backspace and delete character

* - Designates unlabed function key in upper right hand corner of keyboard.

8.4.1.7 CONFIGURATOR ERRORS

The Configurator will display error codes to indicate any errors which occur during user prompt response or during Configurator execution. All errors are reported in two basic formats.

The first format displays errors detected by the Configurator or by the Configurator/TX990 interface. Many of these errors will result when an invalid device/file access is attempted in response to a Configurator prompt. The remainder of these errors are detected by the "editor" function of the Configurator during POWER BASIC statement translation into internal pseudocode. The Configurator reports all errors of this type to the user as a decimal error code using the format following format:

```
*ERROR XX
```

where,

XX is the decimal code corresponding to the error which occurred

For example:

```
*ERROR 44
*ERROR 07
```

The error codes and corresponding error messages generated by the Configurator are presented in Appendix A-1 and A-2. Appendix A-1 presents the error codes and messages which apply to both the Host POWER BASIC Interpreter and the Configurator, while Appendix A-2 presents all errors exclusively reported by the Configurator.

The second format displays all errors reported by the TX990 Operating System. These errors are reported to the user as hexadecimal codes with a trailing "H" character using the following format:

```
FILE I/O ERROR 00XXH
```

where,

XX is the hexadecimal code corresponding to the TX990 error which occurred.

Appendix A-3 presents the hexadecimal codes and corresponding error messages reported by the TX990 Operating System.

8.4.2 CONFIGURATOR OPERATION

When the user enters a responseto the final LIST FILE= prompt, the Configurator begins execution. The Configurator first reads the application program from the specified APPLICATION SRCE= device/file. As the Configurator reads each POWER BASIC statement, the statement is

first translated into internal pseudocode by the "editor" or translator function of the Configurator, and then stored into memory. This continues until the entire application program is stored in memory in pseudocode form, and the APPLICATION SRCE= device/file is then closed.

For example:

```
FILE I/O ERROR 0027H
FILE I/O ERROR 0022H
```

If any invalid or unrecognizable statements or functions are encountered during translation, the corresponding error code and the offending statement are displayed on the terminal device and output to the LIST FILE. The entire statement line is omitted from the pseudocode and translation continues with the next statement of the application. The error codes generated by the "editor" are presented in Appendix A-1. The user should carefully look over the detected errors and the original application program to make sure that these statements are not required for application execution and also that they are not referenced by other statements within the program. These type of errors are generally caused by including statements or functions which are only supported by the Host POWER BASIC Interpreter and not by the Configurator or Target POWER BASIC Interpreter.

The Configurator then scans the pseudocode of the application program and determines which object modules need to be included from the object module library in the final Target Interpreter/Application. As each statement and function is encountered in the application program, it is appropriately marked in the "STATEMENTS USED" and "FUNCTIONS USED" tables for later output to the LIST FILE. Similarly, as each object module is referenced by the application program it is marked in the "MODULES USED" table, also for output to the LIST FILE.

Next the OBJECT FILE is opened, and the pseudocode of the application program (or ROOT module) is written to the specified device/file, and the OBJECT FILE is then closed.

The LINK CONTROL device/file is then opened, the link control sequence corresponding to the application program is written, and the specified device/file closed.

The Configurator outputs the listing of the configuration process and closes the specified device/file to complete the configuration of the current application program.

The Configurator then prompts the user for the next program to be configured by output of the APPLICATION SRCE= prompt. The user may enter responses to the next sequence of prompts, or may terminate the Configurator by entering the ESCAPE key (RESET function key on VDT913, or unlabeled function key in upper right-hand corner of VDT911). Also

note that the user may enter the ESCAPE key in reponse to any of the prompts, or during the actual configuration process to terminate Configurator execution. When the Configurator is terminated, all files are closed, and control returns to the TX990 Operating System.

8.4.3 CONFIGURATOR OUTPUTS

The following paragraphs will explain the OBJECT FILE, LINK CONTROL, and LIST FILE Configurator outputs.

8.4.3.1 OBJECT FILE OUTPUT

The OBJECT FILE output (ROOT module) of the configurator will contain the pseudocode representation of the application program. This pseudocode file will generally be used in the generation of a Target POWER BASIC application; however, it may also be programmed directly into EPROM and used as the application program in a Development BASIC (TM990/451) system. Each of these uses will be presented below.

The OBJECT FILE is typically used in the generation of a Target POWER BASIC Interpreter/Application. The device/file specified in response to the OBJECT FILE prompt will receive the pseudocode form of the application program, and an INCLUDE command referencing this module will also be generated in the LINK CONTROL. Execution of the TX990 Link Editor using this Link Control file will then appropriately "INCLUDE" the pseudocode application program into the final Target POWER BASIC Interpreter/Application.

The OBJECT FILE produced by the Configurator has been designed so that this pseudocode form of the application program may be directly programmed into EPROM, placed in the memory map of a Develoment BASIC system (TM990/451) and executed in the desired application. This enables the user to develop the application program on the Host POWER BASIC Interpreter on the FS990, and then transport the pseudocode form of this program directly to TM990 board system without performing the entire configuration process when a customized Target POWER BASIC Interpreter is not required. The OBJECT FILE produced by the Configurator is completely relocatable, and may reside in EPROM anywhere within the memory map of the POWER BASIC system. Once the OBJECT FILE is programmed into EPROM (typically using the TXPROM utility), the user will select the address space in the TM990 system most convenient for its location.

NOTE

The OBJECT FILE produced by the Configurator has the "RUN" option ENABLED, so that if the EPROM's are placed at hexadecimal location 3000_{16}, the application program will immediately begin execution when the RESET switch of the microcomputer board is activated. If the EPROM's are placed at any location other than hex 3000_{16}, activation of the RESET switch will initialize Development BASIC to the keyboard mode awaiting user keyboard input.

If the auto-run option described above is disabled by placing the EPROM's at a memory a dress other than hex 3000_{16}, the "LOAD address" command of Development BASIC (where the address corresponds to the location in memory at which the EPROM's were placed) will initialize BASIC to enable access to the program in EPROM. The user may LIST and execute this program; however, attempts to edit the program will result in an error. For additional information on Development BASIC, and the ability to access application programs in EPROM, refer to the TM990 POWER BASIC REFERENCE MANUAL, MP308.

8.4.3.2 LINK CONTROL FILE OUTPUT

The LINK CONTROL file produced by the Configurator will be used when performing the final link in the generation of a Target POWER BASIC Interpreter/Application. It will contain the appropriate LIBRARY, END, TASK, and INCLUDE commands required to perform the link. The following example shows a typical link control file which the Configurator would generate for an application program.

Example:

```
NOSYMT                      No symbol tables in final object
NOAUTO                      Inhibits automatic external reference resolution
LIBRARY :CBASIC/LIB         Defines sequential object library
TASK CBASIC                 Defines phase of link edit structure
INCLUDE :STARTC/OBJ         Includes core module of Interpreter
INCLUDE (BAUDC)
INCLUDE (CVBDC)
INCLUDE (CVDB)              INCLUDEs object modules from library
INCLUDE (CVGC)              referenced by application program
INCLUDE (PRINTC)
INCLUDE (ZZENDC)
INCLUDE DSC2:PROCES/OBJ     INCLUDEs pseudocode or "ROOT module"
INCLUDE DSC2:ASSEM/OBJ      INCLUDEs CALLed assembly language modules
END                         Defines end of link edit
```

All Configurator generated Link Control files will: 1) specify no symbol tables in the final object module, 2) define the library containing the object modules of the Target POWER BASIC Interpreter, 3) define the link edit as a final task with a name of CBASIC, 4) include the required STARTC/OBJ module, all referenced object modules from the library, and finally the pseudocode representation of the application program (or ROOT module) and any assembly language routines CALLed within the application.

The user should note the following:

- The LIBRARY containing the object modules for the Target POWER BASIC Interpreter must reside on the same diskette drive as the Configurator (ie., it must reside on the default diskette, typically the left hand drive)

- The STARTC/OBJ file must also reside on the same diskette drive as the Configurator (i.e., it must reside on the default diskette, typically the left-hand drive.

- The pseudocode representation of the application program or "ROOT module" should reside on the secondary or application diskette, typically the right-hand drive.

- Any assembly language routines CALLed from the application program must reside on the secondary or application diskette termed DSC2, typically the right-hand drive.

8.4.3.3 LIST FILE OUTPUT

The LIST FILE will contain complete information on the execution of the Configurator. The following paragraphs and examples will explain the contents of this file.

A header will appear at the top of each page of the LIST FILE denoting the release and revision number as follows:

```
POWER BASIC CONFIGURATOR REV C.n.m
```

where,

n = the release number
m = the revision number

The first page of the LIST FILE will contain the Configurator prompts and user responses followed by any "edit" errors which occurred during BASIC statement translation into internal pseudocode. Any errors will be reported by an error code followed by the offending statemen. The error codes generated by the "editor" are presented in Appendix A-1. Immediately following the error report on the first page will be the "NUMBER OF BYTES OF PSEUDO SOURCE=>XXXX" output. This hexadecimal value represents the number of bytes of internal pseudocode required for the application. This is the number of bytes the application program will occupy in the Target POWER BASIC Interpreter/Application. The following example shows the first page of a typical output where some edit errors occurred:

```
POWER BASIC CONFIGURATOR REV C.1.4

    APPLICATION SOURCE=DSC2:PROCESS/SRC
          OBJECT FILE=DSC2:PROCESS/OBJ
       LINK CONTROL=DSC2:PROCESS/LNK

    ( EDIT ERRORS, IF ANY, LISTED HERE )

    *ERROR 07
```

```
 70   EQUATE Y1, Y(1)
*ERROR 44
 360   ESCAPE
*ERROR 07
 400 LET I=SGN(X)

NUMBER OF BYTES OF PSEUDO SOURCE=>01AC
```

The next page of the LIST FILE will present the module summary of the object modules required by the application program. Each of these object modules will be referenced as the operand of an INCLUDE command in the Link Control File, and the corresponding module will be included from the object library during the link of the final Target POWER BASIC application. The module summary also presents the number of primary references and secondary references of each required object module. The number of primary references is incremented each time a module is referenced directly by encountering a statement or function (in the application program) which is contained in that module. The number of secondary references is incremented each time that module is referenced by another module within the library. The folowing example shows a typical module summary output:

POWER BASIC CONFIGURATOR REV C.1.4

MODULE SUMMARY
MODULES USED

NAME	PRIMARY REF	SECONDARY REF	NAME	PRIMARY REF	SECONDARY REF
BAUDC	1	4	CVBDC	0	1
CVDB	0	2	CVGC	0	3
GOSUB	2	0	IF	1	0
IPCOM	0	1	JMP	0	1
LET	2	0	MOVE	0	1
PRINTC	1	0	TICF	2	0
TIMEC	3	0			

The next page of the LIST FILE will present a summary of the verbs (or statements) used in the application program. The summary will indicate the number of times each statement was used within the program, and will also indicate the object module in which the statement is located and any required support modules. The following example shows a typical statement summary output:

POWER BASIC CONFIGURATOR REV C.1.4

STATEMENTS USED

NAME	# OF REFS	MODULES	
GOTO	2	GOSUB	
(LET)	2	LET	
PRINT	1	PRINTC	
TIME	1	TIMEC	
BASE	1	BASE	
BAUD	1	BAUDC	
IF	1	IF	
CRF	2	CRUF	BASE

The next and final page of the LIST FILE will present a summary of the functions used in the application program. The summary will indicate the number of times each function was used within the program the object module in which the function is located as well as any required support modules. The following example shows a typical function summary output:

POWER BASIC CONFIGURATOR REV C.1.4

FUNCTIONS USED

NAME	# OF REFS	MODULES	
TIC	2	TICF	TIMEC
SIN	2	SINF	
CRF	1	CRUF	BASE

8.4.4 CONFIGURATOR TERMINATION

When the LIST FILE is output, the Configurator has completed execution on the current application program and will again issue the APPLICATION SRCE= prompt for the next program to be configured. If no additional programs are to be configured, the user will enter the ESCAPE key (RESET function key on the VDT913 or the unlabeled function key in upper rightr hand corner of the VDT911) to terminate the execution of the Configurator. The next step in the configuration process is the execution of the TX990 Link Editor.

8.5 TX990 LINK EDITOR

The Link Editor enables the user to link together all the modules which comprise the Target POWER BASIC Interpreter/Application and resolve the external references of these modules. To permit fast and efficient linking of these modules, the use of a sequential object library was adopted. This library contains the object modules of the

Target POWER BASIC Interpreter organized in the same order as they are INCLUDED by the Link Control File (ie., alphabetically). This minimizes the search time for a module within the library and requires only one pass of the library for module inclusion. The sequential library approach REQUIRES the use of the TXSLNK Link Editor, version 2.3.1 or later (located on the TI-shipped TXDSLE 2250446-1606 **. 2.3.1) to provide the enhanced support of sequential libraries.

The following paragraphs will present the library and object modules required, as well as the execution sequence for the TX990 Link Editor.

8.5.1 REFERENCED OBJECT MODULES

The Configurator generated Link Control File requires that some of the object files be present on specific diskette drives in the system. The Target Interpreter Object Library and the Interpreter core module START/OBJ) are REQUIRED to be on the same diskette with the Link Editor. Any assembly language routines CALLed by the application program MUST reside on the secondary or application diskette with the device name of DSC2. Also the Root module is recommended to be directed to the secondary or application diskette during Configurator execution.

8.5.2 LINK EDITOR EXECUTION

The user will load and execute the Link Editor via the TXDS Control Program. The user will call the TXDS Control Program into execution by performing the steps as outlined in Section 2, "LOADING AND EXECUTION OF HQST POWER BASIC INTERPRETER AND CONFIGURATOR". The TXDS Control Program assists in program loading and execution by displaying the following prompts om the system console:

```
TXDS    936215    *A    291/78    10:30

PROGRAM:
  INPUT:
 OUTPUT:
OPTIONS:
```

The user will respond to these prompts by entering the Link Editor program pathname in response to the PROGRAM prompt, the Link Control File pathname in response to the INPUT prompt, the linked object file pathname in response to the OUTPUT prompt, and the options required to the OPTIONS prompt.

The Link Editor must reside on the primary diskette, typically the left hand drive with a device name of DSC in the standard TI-shipped operating system. Therefore, the following response is entered:

```
PROGRAM:  DSC:TXSLNK/SYS
```

Once this entry is complete, the INPUT prompt is displayed. In response to this prompt, the user enters the pathname of the file or device from which the control file is to be read. The Link Control File is generated by the Configurator, and generally resides on the secondary or application diskette. The device/file which received the Link Control File was specified by the user response to the LINK CONTROL= prompt. This pathname should be entered in response to the INPUT prompt. Examples are:

```
INPUT:  DSC2:FAN/LNK
INPUT:  DSC2:VALVE/LNK
```

Next the OUTPUT prompt is presented, which requests that the following entries be made:

```
OUTPUT: <object>,[load map],[scratch]
```

These parameters have the following meanings:

object - specifies the linked object output pathname

load map - specifies the output listing pathname. If no entry is made, the system default printer is used.

scratch - specifies the diskette unit upon which the scratch files will be created. If no value is entered, the scratch files are created on the same diskette unit from which the Link Editor was loaded.

Typically the user will direct the object output to the secondary or application diskette, will direct the load map to the Line printer and will take the default value for the scratch file parameter. For example:

```
OUTPUT:  DSC2:MOTOR/LOB, LP
OUTPUT:  DSC2:VALVE/LOB, LP
```

Next the OPTIONS prompt will be displayed. In response to this prompt, the user can either enter the number of bytes (in decimal) of memory available to the Link Editor for the table area, or make no entry and accept the default of 8192 bytes. It is recommended that the user enter the largest memory size value which the operating system will accept to insure that enough memory is allocated to successfully link the program. The user will need to perform a trial and error approach to determine this value for the particular operating system configuration being used. Any entry made by the user to the OPTIONS prompt is preceeded by the letter M. For example:

```
OPTIONS:  M15000
OPTIONS:  M12000
```

The following examples show typical responses to the TXDS Control Program prompts:

```
TXDS    936215    *A    291/78    11:00

PROGRAM:DSC:TXSLNK/SYS
  INPUT:DSC2:PROCESS/LNK
 OUTPUT:DSC2:PROCESS/LOB, LP
OPTIONS:M15000
```

or similarly,

```
TXDS    936215    *A    291/78    11:00

PROGRAM:DSC:TXSLNK/SYS
  INPUT:DSC2:MOTOR/LNK
 OUTPUT:DSC2:MOTOR/LOB, LP
OPTIONS:M15000
```

The Link Editor will then begin execution. It will read the control file, include the specified object files, and perform the search of the library for the specified modules to be included. This process will continue until all of the INCLUDES have been processed, with final undefined external reference resolution being performed before Link Editor termination. For additional information on the execution of the Link Editor refer to the "MODEL 990 COMPUTER LINK EDITOR REFERENCE MANUAL", manual number 949617-9701

8.5.3 LINK EDITOR OUTPUT

The listing produced by the Link Editor, using the Configurator generated Link Control File and the execution sequence as outlined above, will contain the following: 1) a listing of the Link Control File, 2) any multiply defined symbols, 3) the link map of the object modules included, and 4) the link map the symbol definitions wthin these modules. Figure 8-2 illustrates a typical listing output of the Link Editor.

Note that all listings produced by the Link Editor will indicate a series of multiply defined symbols encountered during the linking process. This occurs because the final module included from the library (ie., ZZENDC) has externally defined symbols corresponding to the entry points of ALL modules within the library. This prevents final automatic external reference resolution from automatically including ALL remaining modules in the library which were not explicitly INCLUDED, but were referenced by the STARTC/OBJ module. This will also eliminate the listing of unresolved references by the Link Editor. This results in the occurance of the multiply defined symbols in the output listing, but the actual linked object output

will be correct and will have included the correct modules from within the library. (Refer to Figure 8-2 on the following pages).

8.6 TARGET (CONFIGURED) POWER BASIC APPLICATION

Upon completion of Link Editor execution, the specified OUTPUT device/file will contain the final Target POWER BASIC application object module. This object module should first be tested to verify its operation, and may then be programmed in EPROM's and placed in the final TM990 target application. The following paragraphs will present these final two steps in the configuration of a Target POWER BASIC application.

8.6.1 AMPL CHECKOUT PROCEDURES

The user may load the resulting linked object module (Target POWER BASIC application) into the target system RAM using AMPL. The target system must be configured such that there is RAM mapped where EPROM would normally be in the target system (ie., beginning at 0000), and that there is also additional RAM area for the workspace of the interpreter from hexadecimal address $FFFF_{16}$ on down in memory. Note that AMPL emulator memory (EUM) is located at 0000 through $1FFF_{16}$, and can be mapped into the Target memory. Note that any additional RAM required to replace the EPROM address space, must be acquired from expansion memory boards. The workspace area RAM for the application will be acquired from the microcomputer board and any additional expansion memory boards.

The user may initialize the AMPL emulator, configure the emulator and trace memory, and then oad the Target POWER BASIC application using the load program command as shown in the following example:

```
EINT('EMU')
EUM=ON
ETM=OFF
LOAD (':TEST/SYS',0)
```

Note that the bias MUST be zero in order for the target application to be properly loaded.

```
 TXSLNK          2.3.1   78.244          01/00/00   02:27:50
COMMAND LIST

NOSYMT
NOAUTO
LIBRARY :CBASIC/LIB
TASK CBASIC
INCLUDE :START0/OBJ
INCLUDE (CALLC)
INCLUDE (CVDB)
INCLUDE (CVGC)
INCLUDE (LET)
INCLUDE (ZZEND0)
INCLUDE DSC2:TEST5/OBJ
INCLUDE DSC2:ABC/OBJ
INCLUDE DSC2:BCD/OBJ
END


SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = CLLY       MODULES = 2, 6

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = LETY       MODULES = 5, 6


 TXSLNK          2.3.1   78.244          01/00/00   02:27:50
LINK MAP

CONTROL FILE = DSC2:TEST5/LNK

LINKED OUTPUT FILE = DSC2:TEST5/SYS

LIST FILE = LP

OUTPUT FORMAT = ASCII

LIBRARIES

NO    ORGANIZATION    PATHNAME

1     SEQUENTIAL      :CBASIC/LIB
```

FIGURE 8-2. LINK EDITOR. EXAMPLE (Sheet 1 of 3)

PHASE 0, CRASIC ORIGIN = 0000 LENGTH = 0F96

MODULE	NO	ORIGIN	LENGTH	TYPE	DATE	TIME	CREATOR
STARTC	1	0000	0804	INCLUDE	01/02/79	14:08:03	SDSLNK
CALLC	2	0804	0044	INCLUDE,1	01/03/79	09:51:46	SDSLNK
CVDA	3	0848	0152	INCLUDE,1	01/03/79	09:52:51	SDSLNK
CVGC	4	0CFA	005E	INCLUDE,1	01/03/79	09:53:06	SDSLNK
LET	5	0D58	0142	INCLUDE,1	01/03/79	09:56:32	SDSLNK
ZZENDC	6	0E9A	000E	INCLUDE,1	01/03/79	10:01:02	SDSLNK
ROOT	7	0EA8	00E2	INCLUDE			CONFIG
ABC	8	0F8A	000A	INCLUDE	11/21/78	12:14:22	SDSMAC
BCD	9	0F94	0002	INCLUDE	12/18/78	12:53:26	SDSMAC

D E F I N I T I O N S

NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO
ABC	0F8A	8	ABSF	0F9A	6	*ADDF	0964	1	ANDF	0E9A	6
ASCF	0F9A	6	*ASR0	003C*	1	*ASR1	003E*	1	ATNF	0E9A	6
*B01	0961	1	*B05	0965	1	*B20	01FE	1	*B3F	01FF	1
BASY	0E9A	6	BAUD	0E9A	6	BCD	0F94	9	*BCRU	003A*	1
BITF	0E9A	6	BITY	0E9A	6	*BUS	000C*	1	*C1	0960	1
*C4	0962	1	*C4A00	01FC	1	*C6	096A	1	*CCNT	002A*	1
CKEX	0586	1	*CLKADR	FE6C*	1	CLKI	0E9E	6	*CLKT01	FE72*	1
*CLKT02	FE74*	1	*CLKWS	FE6A*	1	CLLTAB	0F6A	7	CLLY	0804	2
COSF	0E9A	6	CRBF	0E9A	6	CRBY	0F9A	6	CRFF	0F9A	6
CRFY	0E9A	6	CVBD	0E9C	6	CVBI	0F9C	6	CVC10	0D22	4
CVCH	FEEC*	1	*CVDB20	0C82	3	CVDIFZ	0A4C	3	*CVDIZ	0B4A	3
*CVGCN	0CFA	4	CVGCN1	0D02	4	*CVHD	FEF2*	1	*CVHD01	FEF3*	1
*CVHD15	FF01*	1	*DATXB	0201	1	*DDM	0022*	1	DEFY	0E9A	6
DIMY	0F9A	6	*DLC	0020*	1	*DLIM	001F*	1	*DS	FE76*	1
*DS1	FE7C*	1	*DS2	FF8C*	1	*DSS	FFC2*	1	*EBP	FFF2*	1
*EFLG	002A*	1	*ELNM	002E*	1	*ELSF	0030*	1	*ELSY	0F9A	6
*ENUM	002C*	1	ERRY	0F9A	6	*EUS	0002*	1	*EVAL	073C	1
*EVALS2	0734	1	EVARZ	0582	1	EVERZ	05CC	1	*EVFX	05E4	1
*EVOP3A	087E	1	EVSDZ	0574	1	*EVSFR	0714	1	*EVSKB	FEF2*	1
*EVSKE	FF94*	1	EXPF	0F9A	6	*FAD	0246	1	*FADDI	051E	1
*FCLP	044E	1	*FDD	0482	1	FFLG	0037*	1	*FIX	0AB2	1
*FLDD	0234	1	*FLOAT	0300	1	*FMD	0362	1	*FNEG	02F6	1
*FNRM	032A	1	*FNS	0004*	1	*FNSZ	000A*	1	FORY	0E9A	6
*FPAC	FF9C*	1	*FPAC2	FF9E*	1	*FPAC4	FFA0*	1	*FPWP	FF9C*	1
*FSCL	0546	1	*FSD	023C	1	*FSRD	022C	1	*FSUBI	0532	1
*FUZZ	3800*	1	GOS1	0FA2	6	GOSR1	0FA2	6	GOSY	0E9A	6
GOTY	0E9A	6	*GSC	C01A*	1	*GSS	0006*	1	*GSSZ	0014*	1
*HFLG	0034*	1	*IFLG	0036*	1	IFY	0F9A	6	IMKY	0F9A	6
INPF	0E9A	6	INPY	0F9A	6	*INTFLG	FF56*	1	INITW	0E9E	6
*INTSB	FE36*	1	*INTWP2	FF48*	1	IOM	0000*	1			
IRTY	0E9A	6	*ISTCK	FF36*	1	*ITAB	FE1E*	1	*ITAB2	FE18*	1
LANDF	0E9A	6	*LEC	0024*	1	LENF	0F9A	6	LETY	0D5A	5
*LINE	0160	1	*LINE0	016F	1	*LINE2	0178	1	*LINE5	01A0	1
LNOTF	0E9A	6	*LNSZ	0084*	1	LOGF	0F9A	6	LORF	0E9A	6
LXORF	0E9A	6	MCRF	0E9A	6	MEMF	0F9A	6	MEMY	0E9A	6
*MODE	0040*	1	*MIC	0064*	1	MKYF	0F9A	6	NLIN	015A	1

FIGURE 8-2. LINK EDITOR EXAMPLE (Sheet 2 of 3)

```
 TXSLNK          2.3.1  78.244          01/06/00  00:12:28                PAGE   4
 NAME    VALUE NO    NAME    VALUE NO    NAME    VALUE NO    NAME    VALUE NO

*NLINO   0156  1     NOTF    0F9A  6     *NRV    0004* 1     *NVO    0012* 1
*NVS     0014* 1     *NXTXB  0200  1      NXTY   0F9A  6      ONY    0F9A  6
 ORF     0E9A  6     *PFIX   0ADE  1     *PLC    0C1C* 1     *PLF    003A* 1
 POPY    0E9A  6      PRWF   0E9A  6      PRTY   0E9A  6     *RANDS  FE68* 1
 RANDZ   0EA4  6      RANY   0E9A  6      RDDY   0F9A  6     *REST   0134  1
 RNWY    0E9A  6      ROOT   0FAA  7      RTNY   0F9A  6     *SFSN   0AF0  1
 SINF    0E9A  6     *SLN    0026* 1     *SLT    000E* 1      SORF   0E9A  6
*SORI    0004* 1      SPHF   0F9A  6     *SSP    FF76* 1     *SUHF   090C  1
 SYSF    0E9A  6     *TFMB   FF9A* 1     *TEMP2  FF96* 1     *TEMP4  FF98* 1
*TEMP6   FF9A* 1      TICF   0E9A  6      TTNY   0E9A  6      TRPY   0E9A  6
*TYO     0018* 1     *UFT    0006* 1     *UMSK   0003* 1     *UNIT   001A* 1
 UNTY    0E9A  6     *VDT    0008* 1     *VNT    0010* 1      WPR1   FFAA* 1
 WPR103  FE9D* 1      WPR104 FE9F* 1      WPR108 FF9A* 1      WPR2   FFDC* 1


2 WARNINGS

**** LINKING COMPLETED
```

FIGURE 8-2. LINK EDITOR EXAMPLE (Sheet 3 of 3)

The user will now simulate the RESET function by setting the workspace pointer (WP) and program counter (PC) to the values contained in the RESET vector. For example:

```
WP=@0
PC=@2
```

The user may now execute the application by entering the start emulator command. For example:

```
ERUN
```

The users application should perform as it did during final system test. In the event it does not, the user should request emulator status via the EST command. If the status indicated that an IDLE instruction is being executed, a fatal error has occurred in the POWER BASIC Interpreter. In the event this occurs, the user should halt the emulator (via the EHLT command) and display the workspace registers via the DR command. The user may obtain the error code from register 0 and the statement number where the error occurred from register 1. These values are displayed in hexadecimal and may assist the user in determining corective action for the error. Any errors detected in the application program will then be corrected in the original application of the Host POWER BASIC Interpreter and the configuration process must be executed again. Note that the execution of a STOP or END statement in the Application program will cause execution of the IDLE statement. However, the error code and line number do not apply.

If the application does not function correctly and a fatal error is not indicated, the user should verify the correctness of the memory map, and the correct function of the target application hardware. The target memory may be verified using AMPL target memory display statements. Application hardware cperation may be partially verified through the use of the CRU instructions supported by AMPL. If the target memory map is correct and the application hardware functions correctly, the user should return to the Host POWER BASIC Interpreter and attempt to debug the application interactively. If the application functions correctly on the Host POWER BASIC Interpreter but does not operate correctly in the "configured" target application, the user should contact the local TI distributor.

8.6.2 PROGRAMMING USING TXPROM UTILITY

The Target POWER BASIC application may then be programmmed into EPROM using the TXDS prom programmer utility program (TXPROM). To use this utility, the user must recall: 1) that the resultant object module (Target POWER BASIC application) produced by the Link Editor is relocatable, and 2) that the Target POWER BASIC application must reside in the TM990 system beginning at location 0000.

The user must define the PROGRAM and VERIFY control files used by the prom programmer utility to reference the resultant linked oject module. The user will also typically define the control files to program the application into TMS2716 EPROM's. As the user programs each EPROM, it must be appropriately marked as the high or low order byte and with appropriate memory address location. The user may also mark them with the U-numbers (U42-U45) as they are to be placed on the microcomputer board.

For complete information on the use of the TXPROM utility, refer to Section 10 of the MODEL 990 COMPUTER TERMINAL EXECUTIVE DEVELOPMENT SYSTEM PROGRAMMER'S GUIDE entitled "TXDS PROM (TXPROM) Programmer Utility Program".

8.7 EXAMPLES

The user should refer to Appendix E for examples of the configuration process. These examples will illustrate the entire configuration procedure. From the application program "SAVEd" in a diskette file by the Host POWER BASIC Interpreter, to execution of the final Target POWER BASIC application.

SECTION IX

INSTALLATION OF TARGET POWER BASIC APPLICATION

9.1 INTRODUCTION

Once the user has programmed the Target POWER BASIC Interpreter into EPROM (using the procedures as outlined in Section 8, paragraph 8.5.2), the user will need to insert these EPROM's into the Microcomputer board, as well as appropriately configure the TM990 system for use with the final Target POWER BASIC application. This section will present the EPROM installation procedures and the initial system set-up for use in a Target POWER BASIC application. Note that only the general system requirements for the execution of the Target POWER BASIC Interpreter in a TM990 system will be specified, and that any additional application dependent board configurations must be performed by the user.

9.2 GENERAL SYSTEM CONFIGURATION

The Target POWER BASIC application will reside in EPROM starting at the address of 0000 and will occupy as many EPROM's as required to store the interpreter and application. The EPROM's must be inserted in the system at the starting address of 0000_{16} since POWER BASIC utilizes the XOP's and the interrupt vectors. These EPROM's will reside on the TM990/101M or TM990/100M Microcomputer board. If more than four EPROM's are required, the user will place those remaining on a TM990/201 Expansion Memory Board.

The Target POWER BASIC application also requires some RAM area in the TM990 system for Interpreter overhead, and for storage of variables and arrays referenced by the application program. The Target Interpreter requires that the RAM area begins at hexadecimal address FFFF and expands on down in memory (ie., toward address 0000). The TM990/101M and TM990/100M Microcomputer boards (when fully populated) contain RAM from $FFFF_{16}$ down to $F000_{16}$ (4k bytes) and from $FFFF_{16}$ down to $FC00_{16}$(1K bytes), respectively. In many applications, the on-board RAM of the TM990/101M Microcomputer will fulfill the system requirements. However, additional expansion RAM is required in all TM990/100M board systems, or when large applications are "configured" for execution in a TM990/101M system. In both cases, a TM990/201 or TM990/206 Expansion Memory Board will be used to expand the RAM area from hexadecimal address $EFFF_{16}$ on down. Note that the Target POWER BASIC Interpreter will appropriately comprehend and compensate for the resulting "hole" in RAM (from $F000_{16}$ up to $FBFF_{16}$) when a TM990/100M board is used in the application system with an expansion memory board.

9.3 EQUIPMENT REQUIREMENTS

The required equipment and options of the Target application system are described in the following paragraphs.

9.3.1 MICROCOMPUTER BOARD

One of the following microcomputer boards will be required:

- TM990/101M (-1, -2, -3, or -10) Microcomputer Board
- TM990/100M (-1, -2, or -3) Microcomputer Board

9.3.2 OPTIONAL BOARDS

The user may optionally configure any of the following boards in the system:

- TM990/201 (-41, -42, or -43) Expansion Memory Board
- TM990/206 (-41 or -42) Expansion Memory Board
- TM990/310 I/O Expansion Board

In addition, the user may configure any special purpose, application dependent boards into the system as required.

9.3.3 POWER SUPPLY

The power requirements of the boards that are typically used in system configurations are listed in Table 9-1. The power supply must be capable of supplying the total required current and voltage levels of the selected system configuration as a minimum. It is recommended that current ratings of the power supply be specified above the minimum to avoid prolonged use of the power supply at or near its rated maximum. Regulation must be 3% on all supplies.

9.3.4 CHASSIS

The use of a TM990/510 4 slot chassis, TM990/520 8 slot chassis, or equivalent is necessary for the set-up of the Target POWER BASIC application, since typically more than one board is required in most applications.

Alternately, the following 100-pin, 0.125 inch (center-to center) PCB edge connectors may be used to interface with connector P1, such as wire wrap models:

- TI H321150
- AMPHENOL 225-804-504
- VIKING 3VH50/9CND5
- ELCO 00-6064-100-061-001

9.3.5 TERMINAL AND CABLES

The Target POWER BASIC application may include communications (either keyboard input or printer output) with a terminal device. If a terminal device is not required in the particular application, the user may proceed to the next paragraph.

The Target POWER BASIC Interpreter will support the following types of terminal devices:

- RS-232-C compatible terminal (using a TM990/502 cable), or the TI ASR 733 (using a TM990/505 cable): see Appendix B of the 'TM990/100M' or 'TM990/101M Microcomputer User's Guide' to verify the cabling you have or for instructions to make a custom cable.

- TI 743/745: See Appendix B of the 'TM990/100M' or 'TM990/101M Microcomputer User's Guide' for special cabling requirements. The TM990/503 cable may be used to interface a 743/745 terminal.

- Teletype Model 3320/5JE (for TM990/100M-1, TM990/101M-1, and -3 microcomputer boards only): see Appendix A of the 'TM990/100M' or 'TM990/101M Microcomputer User's Guide' for required modifications for 20 mA neutral current loop operation and proper cable connections to the TM990/504 cable.

TABLE 9-1
TM990 MICROCOMPUTER FAMILY POWER CONSUMPTION

TM990 CIRCUIT BOARD	CURRENT REQUIREMENTS					
	5V		12V		-12V	
	MAX	TYP	MAX	TYP	MAX	TYP
/100M CPU BOARD	1.4	1.0	.3	.2	.3	.2
/101M CPU BOARD	2.6	1.2	.5	.25	.4	.2
/201-41 MEMORY BOARD	2.5	1.0	.18	.16	.5	.05
/201-42 MEMORY BOARD	3.0	1.4	.38	.23	.55	.13
/201-43 MEMORY BOARD	5.5	2.15	.75	.48	.7	.23
/206-41 MEMORY BOARD	2.5	1.3	X	X	X	X
/206-42 MEMORY BOARD	5.5	2.15	X	X	X	X

A 25-pin RS-232 male plug, type DB25P, is required if an interface cable is not purchased.

NOTE

POWER BASIC requires use of a standard USASCII coded terminal device. Most terminals use this standard character code. Also, be sure that the correct cable assembly is used with your data terminal. For teletypewriters (TTY), refer to Appendix A of the 'TM990/100M' or 'TM990/101M Microcomputer User's Guide' for the signal configuration used by the main I/O port.

9.4 SYSTEM SETUP

This section describes the steps required to set-up the system for execution of the Target POWER BASIC application.

9.4.1 POWER SUPPLY CONNECTIONS

Figures 9-1 and 9-2 illustrate power supply hookup by connection to a lone 100 pin edge connector and by installation in a card chassis, respectively. Only the TM990/101M microcomputer board is displayed in the figures. However, the figures are applicable to both the TM990/100M and the TM990/101M microcomputer boards.

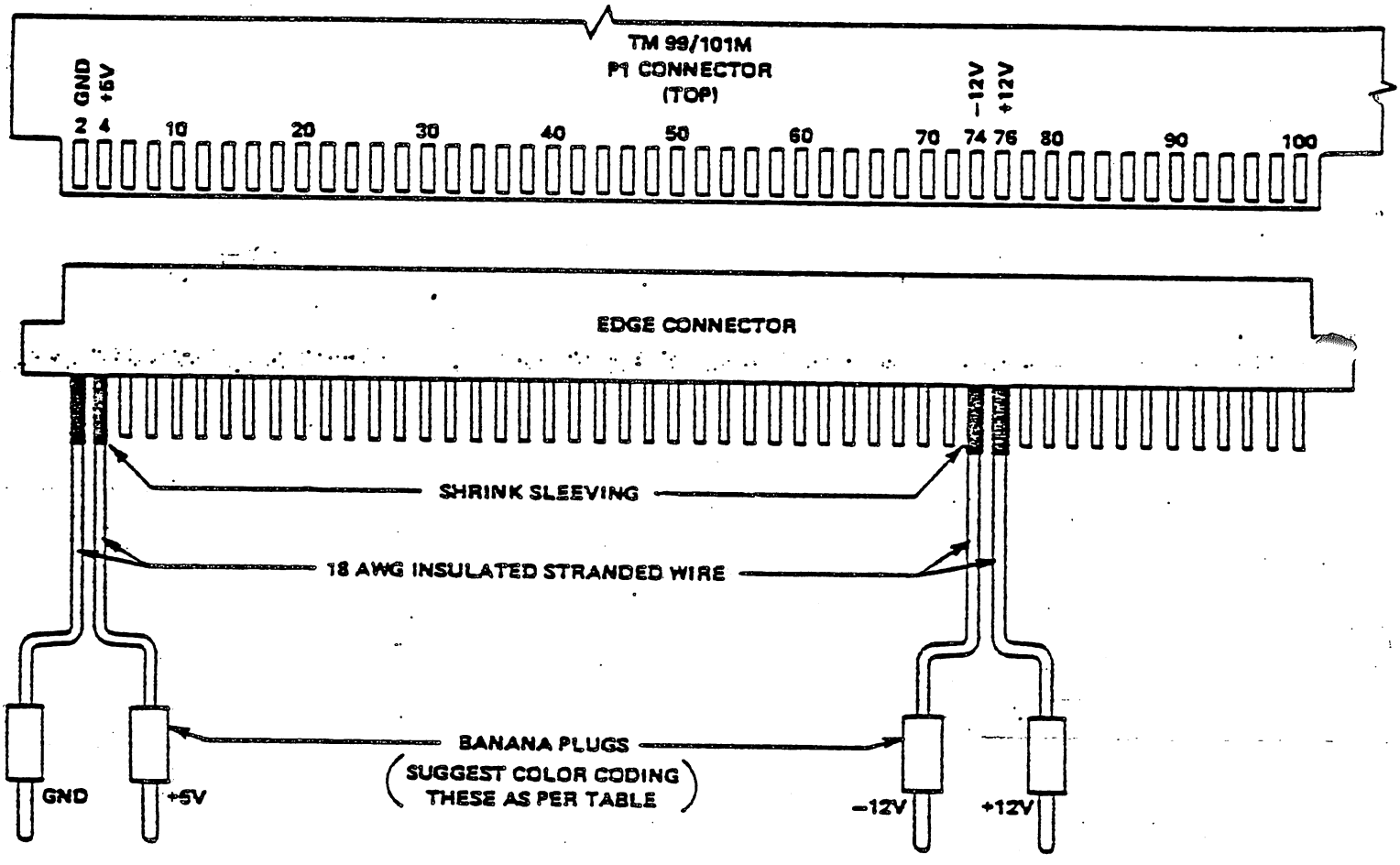
Figure 9-1 shows how the power supply is connected to the microcomputer board through connector P1, using a 100-pin edge connector. Be careful to use the correct pins as numbered on the board; these pin numbers may not necessarily correspond to the numbers on the particular edge connector used. Check connections with an ohmmeter before applying power if there is any doubt about the quality or location of a connection.

The table in Figure 9-1 shows suggested color coding for the power supply plugs. To prevent incorrect connection, label the topside of the edge connector "TOP" and the bottom "TURN OVER".

For power connection to one of the chassis, look at the backside of the backplane, find the connections for each of the supply voltages and connect them to the power supply. Be sure to turn power off before installing or removing any boards from the chassis.

CAUTION

BEFORE connecting the power supply to the microcomputer, use a volt-ohmmeter to verify that correct voltages are present at the power supply. After verification, switch the power supply OFF, and then make the connections to the chassis as shown in Figure 9-2. The correct voltages should also be verified at the chassis or edge connector prior to insertion of a board.



VOLTAGE	P1 PIN*	SUGGESTED PLUG COLORS
+5V	3, 4, 97, 98	RED
+12V	75, 76	BLUE
-12V	73, 74	GREEN
GND	1, 2, 99, 100	BLACK

*ON BOARD, ODD-NUMBERED PADS ARE DIRECTLY BENEATH EVEN-NUMBERED PADS.

A0001417

FIGURE 9-1. POWER SUPPLY HOOKUP

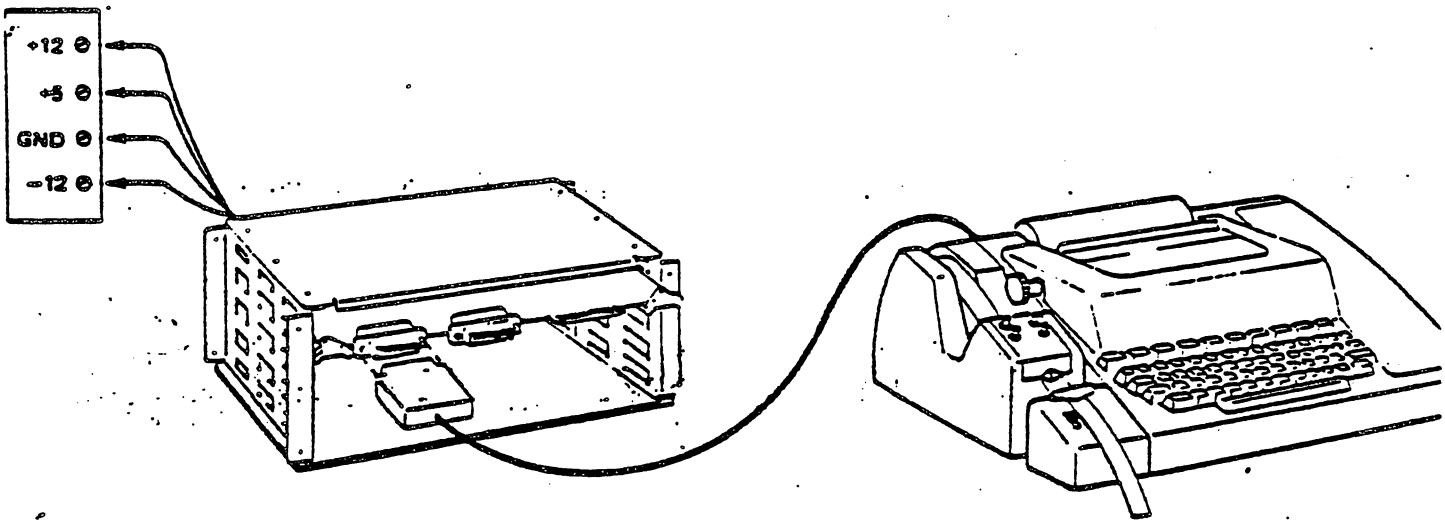


FIGURE 9-2. TM990/101M BOARD IN TM990/510 CHASSIS

9.4.2 EPROM INSERTION

The user should carefully remove all EPROM chips from the board (in sockets U42, U43, U44, and U45) if present, and place them in conductive foam for safe keeping. The user should then remove the Target POWER BASIC application EPROM's from their conductive foam one at a time, beginning with the one marked as U42 by the user, and carefully insert them into the appropriate sockets on the microcomputer board. Make sure they are placed in the correct sockets, and that pin 1 of the EPROM's match with pin 1 as marked by the silkscreen on the board. Carefully inspect each EPROM to ensure that all pins have seated correctly into the socket and that none have bent under the device. Be careful to avoid bending the pins at all times. Bent EPROM pins are the number one cause of "mysterious" board failures.

If more than four EPROM's are required, place the additional EPROM's into the appropriate sockets of the TM990/201 expansion memory board. Note that the TM990/201 board must be configured with EPROM starting at hexadecimal address 2000_{16} as shown by the switch settings of Table 9-6. The expansion EPROM's will then be placed in the corresponding sockets of the memory board so that they reside from memory address 2000_{16} through the address corresponding to the last EPROM of the Target application EPROM set. For additional information refer to paragraph 9.4.4, Expansion Memory Board switch settings.

9.4.3 MICROCOMPUTER BOARD JUMPER SETTINGS

The user should appropriately connect and verify that the jumper configurations on the TM990/101M or TM990/100M board are as described in Table 9-2 or Table 9-3, respectively. Figures 9-3 and 9-4 show the board locations of these jumpers for reference.

For the user who requires detailed jumper placement information, refer to Tables 9-4 and 9-5 for the TM990/101M and TM990/100M boards respectively.

TABLE 9-2

TM990/101M JUMPER SETTINGS

Jumper	Comments
E1-E2, E4-E5, E8-E53, E9-E10, E13-E14, E16-E17, E26-E27, E28-E29 E31-E32, E33-E34, E39-E40, E54-E55	Required
E36-E37	Install for TTY Remove for EIA RS-232-C
All other jumpers	Don't care

TABLE 9-3

TM990/100M JUMPER SETTINGS

Jumper	Jumper Setting
J1	P1-18
J2	2716
J3	16
J4	16
J5-J6	Don't care
J7	EIA
J8-J10	Don't care
J11	Install for TTY Remove for EIA RS-232-C
J12-J18	Don't care

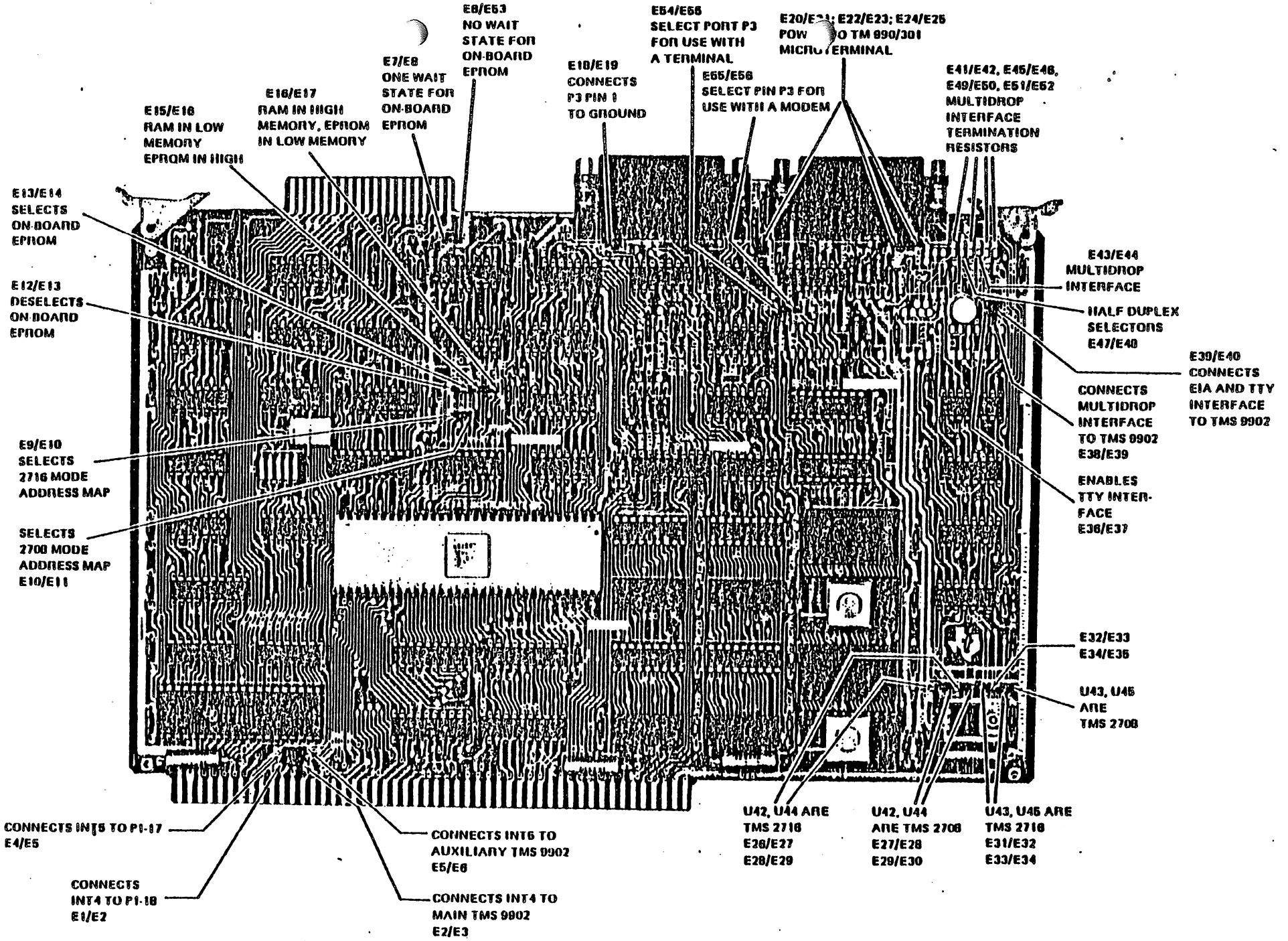


FIGURE 9-3. TM990/101M JUMPER LOCATIONS

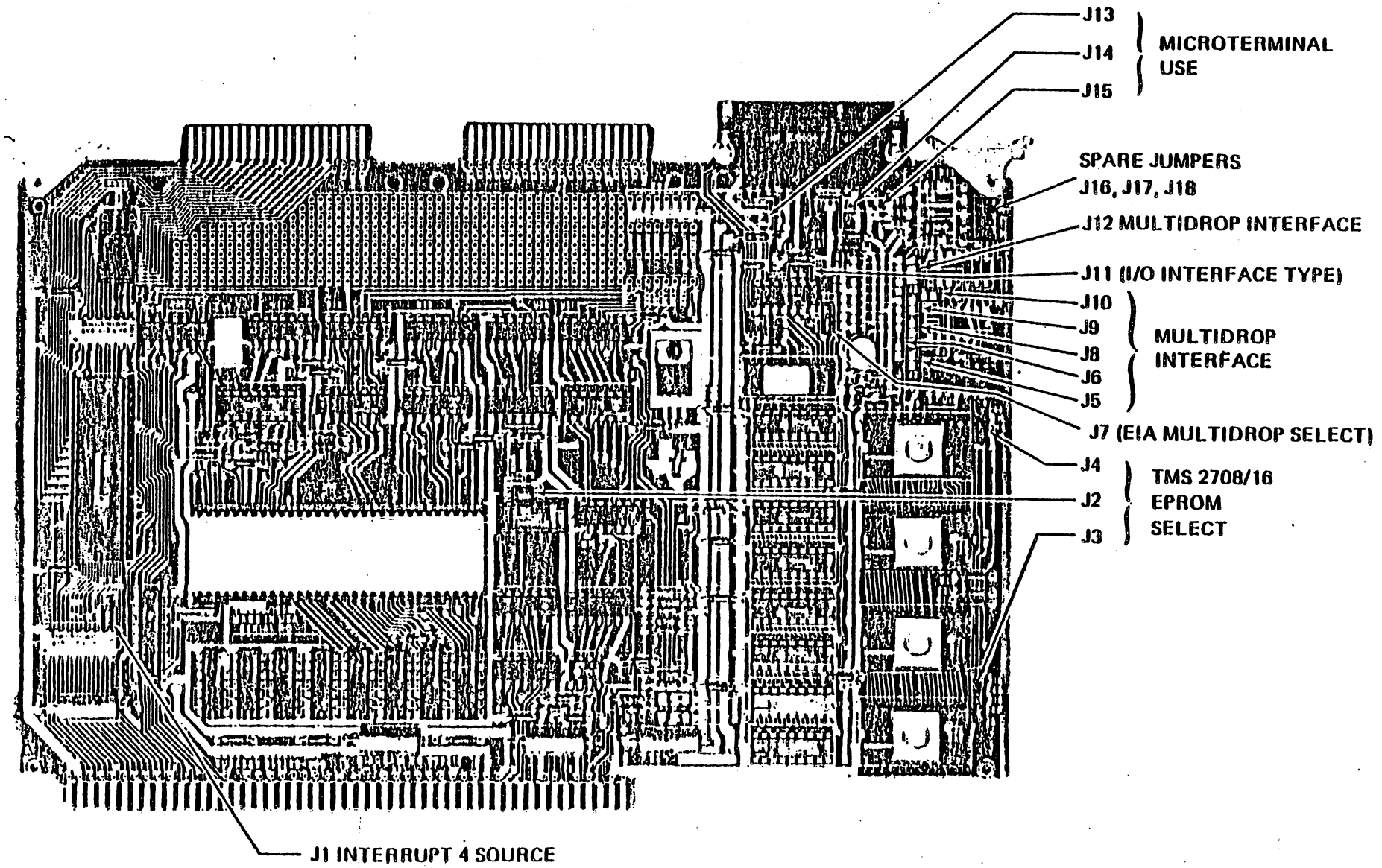


FIGURE 9-4. TM990/100M JUMPER LOCATIONS

TABLE 9-4
TM990/101M BOARD JUMPER POSITIONS

FUNCTION	JUMPER POSITION	EXPLANATION
Interrupt 4 Source	E1-E2	Connects INT 4 to pin 18 of P1 edge connector
Interrupt 5 Source	E4-E5	Connects INT 5 to pin 17 of P1 edge connector
Slow/Fast EPROM	E8-E53	Causes no WAIT states: memory cycles are full speed
2708/2716 Memory Map	E9-E10	Selects memory map for TMS2716 EPROM's
EPROM Enable	E13-E14	On-board EPROM is enabled into memory map
HI/LO Memory Map	E16-E17	EPROM at low address, RAM in high
EIA Connector Ground	E18-E19	Connect PIN 1 of Connector P3 to ground. When using as an auxiliary serial I/O device, consult that devices manual concerning grounding. Normally, this jumper is installed
Microterminal +5 V	E20-E21	Microterminal Power:+5 V to pin 14 of P2 edge connector (See note 1)
Microterminal +12V	E22-E23	Microterminal Power:+12 V to pin 12 of P2 edge connector (see Note 1)
Microterminal -12V	E24-E25	Microterminal Power:-12 V to pin 13 of P2 edge connector (See note 1)
2708/2716 Addressing	E26-E27 E28-E29 E31-E32 E33-E34	Main EPROM is TMS2716 Main EPROM is TMS2716 Expansion EPROM is TMS2716 Expansion EPROM is TMS2716
Teletype*	E36-E37	REMOVE this jumper if using an RS-232 device. If using a teletype device connected to Port 2, INSTALL this jumper
EIA/MD Receive select	E39-E40	This jumper should be INSTALLED when an RS-232 or TTY device is connected to port P2. The multidrop interface is normally not used with POWER BASIC
Multidrop Termination**	E41-E42 E45-E46 E49-E50 E51-E52	These are the connectors for the multidrop termination resistors. These jumpers should be REMOVED since the multidrop interface is normally not used with POWER BASIC
Multidrop Half Duplex**	E43-E44	These jumpers should be REMOVED since multidrop half duplex operation is typecally not required with POWER BASIC.
P3 Port Mode	E54-E55	Connects TMS9902 RTS to CTS for port P3 to communicate with an RS-232 compatible terminal.

* On TM990/101M-1 and -3 only
** On TM990/101M-2 only
Note 1: These jumpers should be removed since the TM990/301 microterminal is not used. (May be left installed for certain terminals, e.g., TI 743.)

TABLE 9-5
TM990/100M BOARD JUMPER POSITIONS

FUNCTION	JUMPER	POSITION	EXPLANATION
Interrupt 4 Source	J1	"P1-18"	Connects INT 4 to pin 18 of P1 edge connection
2708/2716 Memory Map	J2	"2716"	EPROM is TMS2716's
	J3	"16"	
	J4	"16"	
Multidrop Interface	J5	Out	These jumpers should be REMOVED since the multidrop interface is normally not used with POWER BASIC.
	J6	Out	
	J8	Out	
	J9	Out	
	J10	Out	
	J12	Out	
EIA/Multidrop Select	J7	"EIA"	An RS-232 or TTY device is normally connected to the serial port (jumper INSTALLED).
20mA/RS-232 Interface	J11	In/Out	REMOVE this jumper is using an RS-232 device. If using a TTY device, INSTALL this jumper.
Microterminal Power	J13	In/Out	These jumpers should be REMOVED since the TM990/301 microterminal is not used with this system. (May be left installed for certain terminals, e.g., TI 743.)
	J14	In/Out	
	J15	In/Out	
Spare Jumpers	J16	X	Spare jumpers, irrelevant to system operation.
	J17	X	
	J18	X	

X - Don't care

9.4.4 EXPANSION MEMORY BOARD SWITCH SETTINGS

The TM990/201 or TM990/206 expansion memory boards may be configured into the TM990 sytem to provide either or both expansion RAM and/or expansion EPROM for the Target POWER BASIC application. Figures 9-5 and 9-6 present the switch settings and corresponding memory addresses for the RAM and EPROM of the TM990/201 board, while Figure 9-7 presents the switch settings for the RAM of the TM990/206 board. When setting the switches of these boards, the user should be careful not to overlap memory areas on a single board or between boards in the system. The following paragraphs will present the switch settings for configuring RAM and/or EPROM into the Target application system.

The user will configure expansion EPROM into the TM990 system when additional EPROM area is required to physically place the Target POWER BASIC application EPROM set into the system. The EPROM area on the Microcomputer board resides from hexadecimal address 0000 through 1FFF therefore the expansion EPROM area must begin at hexadecimal address 2000 and will continue to the highest address required to store the Target application. The TM990/201 board will be used to provide this additional EPROM area for the system. Table 9-6 presents the DIP switch settings to configure an additional 8K bytes of EPROM into the system as well as an additional 4K, 8K, or 16K bytes of RAM. Note that either the -42 or -43 versions of the memory board may be configured to include up to 32K of EPROM area if required to store the Target Interpreter/application. EPROM area not required in the system is recommended to be disabled to prevent memory area overlap.

The user may expand the RAM area of the system by configuring additional TM990/201 or TM990/206 memory boards into the system. Additional RAM area will provide more user variable storage for the application program resident in the system. Addition of RAM does not require any modification to the Target Interpreter or to the Microcomputer board if configured as described in the preceeding sections. Note that the RAM on the Microcomputer board resides from $F000_{16}$ to $FFFF_{16}$ on the TM990/101M, or from $FC00_{16}$ to $FFFF_{16}$ on the TM990/100M. The target POWER BASIC Interpreter upon initial power-up will automatically size for all available contiguous RAM starting from $FFFF_{16}$ and continuing on down in memory a word at a time until a write/read mismatch is detected. This therefore requires that the top of the expansion RAM be located at hex location $EFFF_{16}$ if the POWER BASIC Interpreter is to locate and use the expansion RAM, and may continue on down in memory as far as required by the user's application. Note that the Target POWER BASIC Interpreter will detect and appropriately compensate for the "hole" in RAM (from $F000_{16}$ up to $FBFF_{16}$) when a TM990/100M board is used in the application system with an expansion memory board. Table 9-7 presents the DIP switch settings to configure additional RAM from either the /201 or/206 board into the system. Note that these switch settings configure all RAM that is available on the particular version of the memory board into the TM990 system.

A0-A3 (HEX)	HEX MEMORY ADDRESS	MICROCOMPUTER MEMORY MAP /100		SWITCH NO.	SWITCH CODES* 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	HEX
				5	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	
				6	ON	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF	
				7	ON	ON	ON	ON	OFF	OFF	OFF	OFF	ON	ON	ON	ON	OFF	OFF	OFF	OFF	
				8	ON	ON	ON	ON	ON	ON	ON	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	
0	0000-0FFF	EPROM										8BLK0									
1	1000-1FFF	EPROM (EXPAN.)																			
2	2000-2FFF											8BLK2									
3	3000-3FFF																				
4	4000-4FFF																				
5	5000-5FFF																				
6	6000-6FFF																			RAM DISABLED	
7	7000-7FFF																				
8	8000-8FFF										6K WORDS (24 4045's)										
9	9000-9FFF																				
A	A000-AFFF					8BLK0													8BLK0		
B	B000-BFFF				8K WORDS (32 4045's)																
C	C000-CFFF													8BLK0							
D	D000-DFFF					8BLK3							4K WORDS (16 4045's)	8BLK1							
E	E000-EFFF	MAPPED I/O																2K WORDS			
F	F000-FFFF	RAM																			

*OFF = 1, ON = 0

FIGURE 9-5. TM990/201 RAM MEMORY CONFIGURATIONS

A0-A3 (HEX)	HEX MEMORY ADDRESS	MICROCOMPUTER MEMORY MAP /100		SWITCH NO.	SWITCH CODES* 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	HEX
				1	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF	
				2	ON	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF	
				3	ON	ON	ON	ON	OFF	OFF	OFF	OFF	ON	ON	ON	ON	OFF	OFF	OFF	OFF	
				4	ON	ON	ON	ON	ON	ON	ON	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	
0	0000-0FFF	EPROM																			
1	1000-1FFF	EPROM (EXPAN.)								8K WORDS (8 2716's)						4K WORDS (4 2716's)					
2	2000-2FFF				16K WORDS (16 2716's)	EBLK7					EBLK7						EBLK7				
3	3000-3FFF																EBLK6				
4	4000-4FFF																				
5	5000-5FFF										EBLK4										
6	6000-6FFF																			EPROM DISABLED	
7	7000-7FFF																				
8	8000-8FFF																				
9	9000-9FFF					EBLK0															
A	A000-AFFF																				
B	B000-BFFF																				
C	C000-CFFF																				
D	D000-DFFF																				
E	E000-EFFF	MAPPED I/O																			
F	F000-FFFF	RAM																			

*OFF = binary "1"
ON = binary "0"

FIGURE 9-6. TM990/201 EPROM MEMORY CONFIGURATIONS

A0-A3 (HEX)	HEX MEMORY ADDRESS	MICROCOMPUTER MEMORY MAP /100		SWITCH NO.
0	0000-0FFF	EPROM		5 ON, 6 ON, 7 ON, 8 ON
1	1000-1FFF	EPROM (EXPAN.)		5 OFF, 6 ON, 7 ON, 8 ON
2	2000-2FFF			5 ON, 6 OFF, 7 ON, 8 ON
3	3000-3FFF			5 OFF, 6 OFF, 7 ON, 8 ON
4	4000-4FFF			5 ON, 6 ON, 7 OFF, 8 ON
5	5000-5FFF			5 OFF, 6 ON, 7 OFF, 8 ON
6	6000-6FFF			5 ON, 6 OFF, 7 OFF, 8 ON
7	7000-7FFF			5 OFF, 6 OFF, 7 OFF, 8 ON
8	8000-8FFF			5 ON, 6 ON, 7 ON, 8 OFF
9	9000-9FFF			5 OFF, 6 ON, 7 ON, 8 OFF
A	A000-AFFF			5 ON, 6 OFF, 7 ON, 8 OFF
B	B000-BFFF			5 OFF, 6 OFF, 7 ON, 8 OFF
C	C000-CFFF			5 ON, 6 ON, 7 OFF, 8 OFF
D	D000-DFFF			5 OFF, 6 ON, 7 OFF, 8 OFF
E	E000-EFFF	MAPPED I/O		5 ON, 6 OFF, 7 OFF, 8 OFF
F	F000-FFFF	RAM		5 OFF, 6 OFF, 7 OFF, 8 OFF

SWITCH CODES*																HEX
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8K WORDS (32 4045's)								4K WORDS (16 4045's)								
	BLK0								BLK0							
									BLK1							
	BLK4															

*OFF = 1, ON = 0

FIGURE 9-7. RAM (ONLY) CONFIGURATION FOR MODEL 990/206

TABLE 9-6

TM990/201 EXPANSION EPROM CONFIGURATIONS

MEMORY BOARD	EXPANSION EPROM K WORDS	EXPANSION RAM K WORDS	SWITCH SETTINGS							
			S1	S2	S3	S4	S5	S6	S7	S8
TM990/201-41	4K x 16	2K x 16	ON	ON	OFF	OFF	OFF	ON	OFF	OFF
TM990/201-42	4K x 16	4K x 16	ON	ON	OFF	OFF	ON	ON	ON	OFF
TM990/201-43	4K x 16	8K x 16	ON	ON	OFF	OFF	ON	ON	ON	ON

TABLE 9-7

RECOMMENDED RAM EXPANSION CONFIGURATIONS

MEMORY BOARD	EXPANSION EPROM K WORDS	EXPANSION RAM K WORDS	SWITCH SETTINGS							
			S1	S2	S3	S4	S5	S6	S7	S8
TM990/201-41 *	4K x 16	2K x 16	OFF	OFF	OFF	OFF	OFF	ON	OFF	OFF
TM990/201-42 *	8K x 16	4K x 16	OFF	OFF	OFF	OFF	ON	ON	ON	OFF
TM990/201-43 *	16K x 16	8K x 16	OFF	OFF	OFF	OFF	ON	ON	ON	ON
TM990/206-41	X	4K x 16	X	X	X	X	ON	OFF	OFF	OFF
TM990/206-42	X	8K x 16	X	X	X	X	OFF	OFF	OFF	ON

X - NOT APPLICABLE

* - The switch settings to disable all expansion EPROM memory from the system are S1-S4 all OFF.

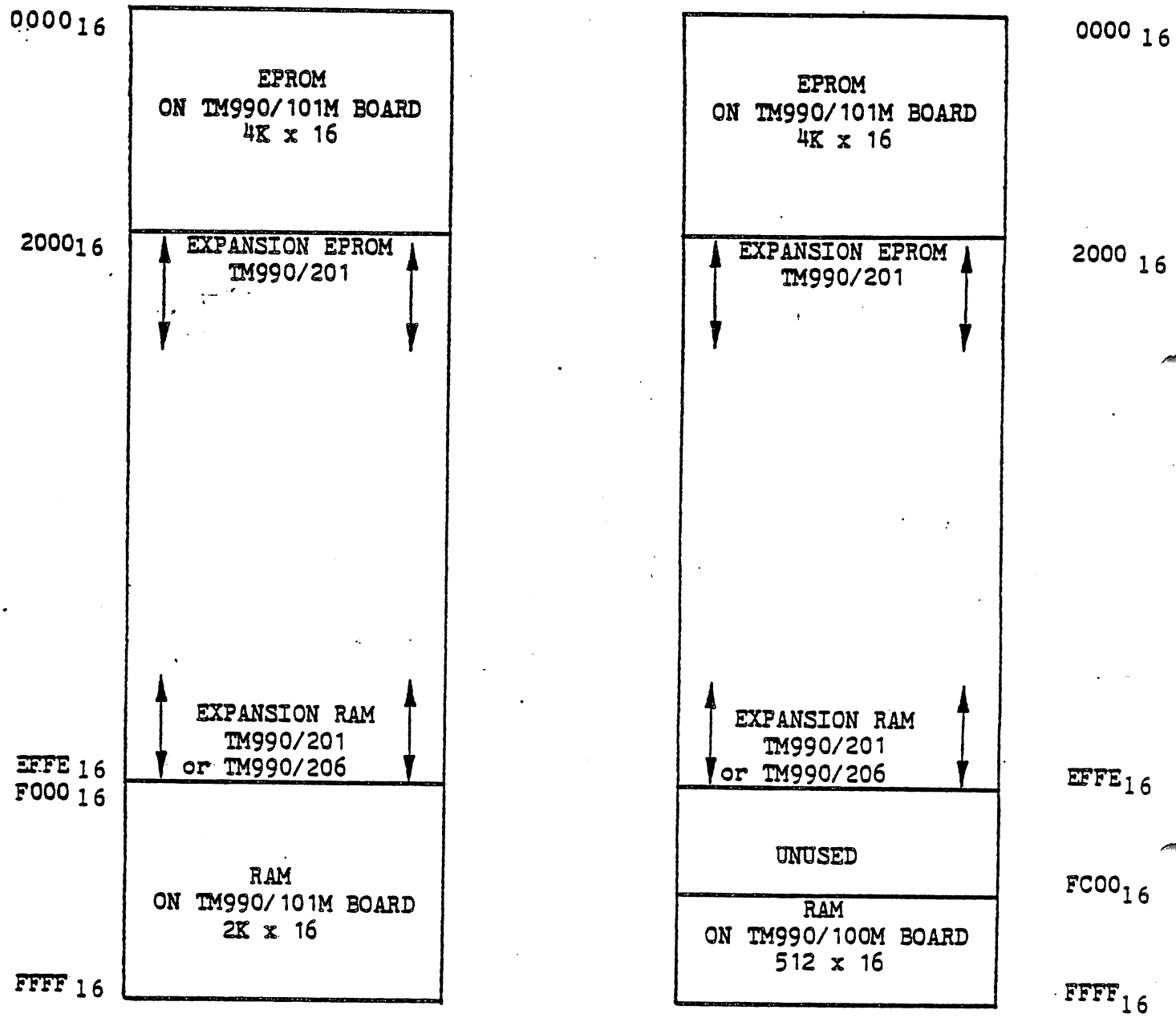


FIGURE 9-8 TARGET POWER BASIC APPLICATION MEMORY MAPS

Figure 9-8 depicts the original memory map of the Target POWER BASIC application, as well as the EPROM and RAM memory expansion areas described earlier in this section.

9.4.5 BOARD INSERTION AND TERMINAL HOOKUP

These procedures assume that the Target POWER BASIC application EPROM's are resident in the required address space as described in paragraphs 9.4.2. Also if the application requires the user of a terminal device, a terminal and cable of the proper type to match the serial interface must be employed (refer to section 9.3.5).

CAUTION

Be very careful to apply the correct voltage levels to the TM990 system. A volt/ohmmeter should be used to verify power supply voltages and connections. Boards should never be inserted in or removed from a system the power applied. This is also true for front edge connections (I/O ports, etc.)

Texas Instruments assumes no responsibility for damage caused by improper wiring or voltage application by the user.

9.4.5.1 BOARD INSERTION

Figure 9-9 shows how to correctly place the microcomputer board in the TM990/510 card chassis. Slot 1 of the chassis is reserved for the microcomputer board because termination resistors for the control bus signals are at the opposite end of the backplane. Slide the microcomputer board into the slot, following the guides. Be sure the microcomputer board P1 connector is correctly aligned in the socket on the backplane, then gently but firmly push the board edge into the edge connector socket.

The second board in a Target POWER BASIC application should be inserted in the next slot down, although this is not critical.

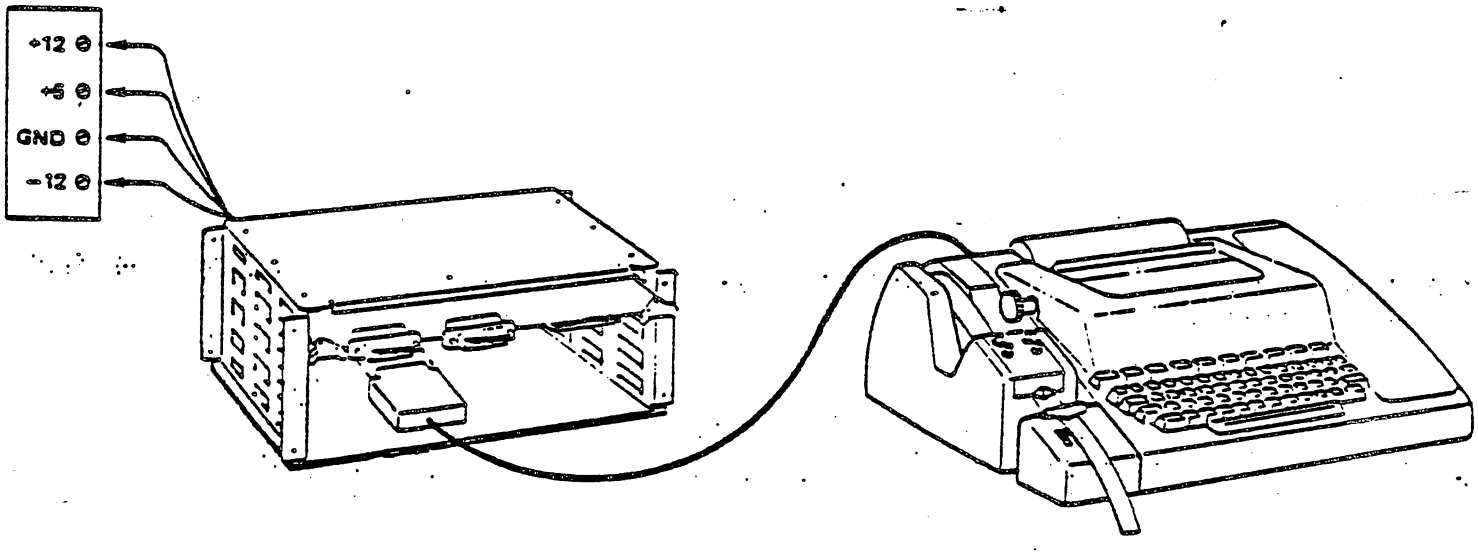


FIGURE 9-9. TM990/101M BOARD IN TM990/510 CHASSIS

9.4.5.2 TERMINAL HOOKUP

If the Target POWER BASIC application utilizes a terminal device for either keyboard input or printer output, the user should perform the following procedures. All keyboard inputs must originate from port A of the TM990/101M Microcomputer board, while any outputs may be directed to either or both ports A and/or B through use of the UNIT statement of POWER BASIC. All keyboard input and/or terminal output will be directed through the single port of the TM990/100M Microcomputer board.

Figure 9-10 shows how the microcomputer board is connected to the TI 743 KSR terminal through connector P2. A DE15S connector attaches to the terminal; a DB 25P connector attaches to P2 on the board. Point-to-point connections between the connectors are shown in the table in Figure 9-10. Figure 9-11 shows a RS-232 terinal (eg., TI 733), and Figure 9-2 shows a TTY, connected to the TM90/101M board through connector P2. All terminals connected to the microcomputer will have a similar hookup procedure and point-to-point configuration. For the differences between terminal cables, refer to Appendices A and B of the 'TM990/101M' or 'TM990/100M Microcomputer User's Guide'.

The target POWER BASIC application may operate the EIA/TTY ports at any of the following baud rates:

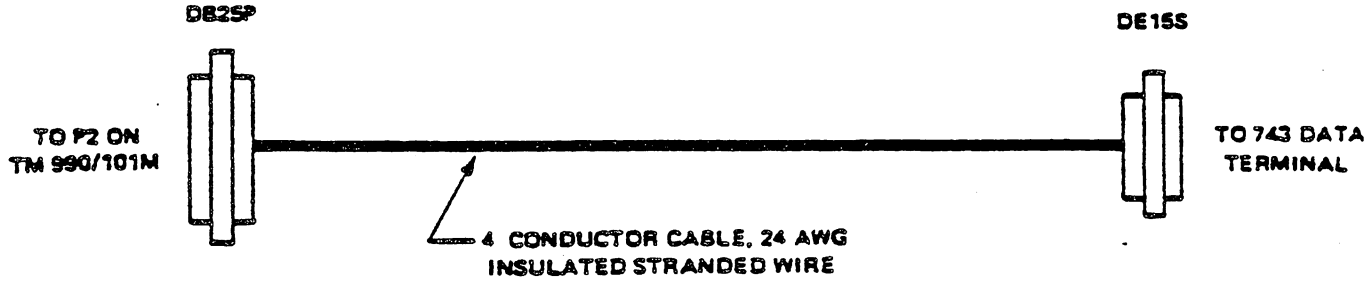
100, 300, 1200, 2400, 4800, 9600, or 19200 baud

To utilize the EIA/TTY port(s) of the microcomputer board, the user must perform initialization of the TMS9902 Asynchronous Communications Controller to the desired baud rate corresponding to the baud rate of the terminal device. The user performs this initialization by including the BAUD statement within the application program before it is "configured" into the Target POWER BASIC application. The user must perform this initialization of port A of the board if it is to be used for keyboard input, and must also perform the initialization on either or both ports A and B if these are to be used for terminal output. The user may then direct all output of the Target application by using the UNIT statement within the application program.

The TMS9902 asynchronous communication controller is initialized for seven-bit ASCII characters, even parity and two stop bits.

The Target POWER BASIC application also uses conversational full-duplex communication. Set the communications mode of your terminal to FULL DUPLEX, and set the OFF/ON LINE switch to ON LINE or the functional equivalents.

Note that there is a 200 ms delay following a carriage return output for all baud rates at or below 1200 baud. This delay allows for



CONNECTIONS		
PIN ON DE15S	PIN ON DB25P	SIGNAL
13	2	XMIT
12	3	RECV
11	8	DCD
1	7	GND

A0001418

FIGURE 9-10. 743 KSR TERMINAL HOOKUP

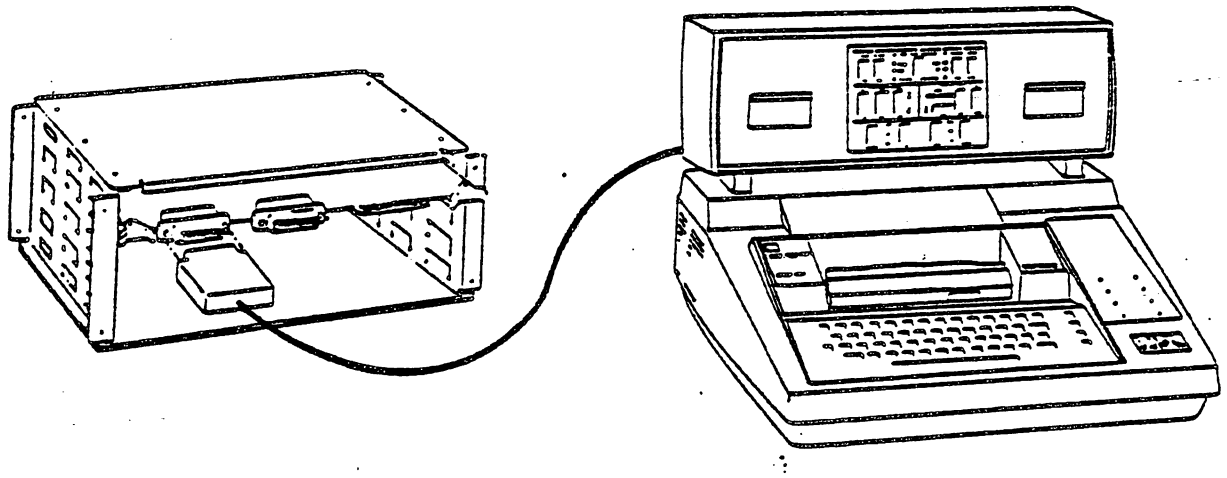


FIGURE 9-11. CONNECTOR P2 CONNECTED TO RE-232-C DEVICE (MODEL 722 ASR)

printhead travel. Also note that 1200 baud output is effectively converted to simulate 300 baud output by insertion of three character wait times between each character output. This occurs on all terminals operating at 1200 baud. This enables proper terminal printer response and prevents thermal printhead overheating when using the TI Silent 700 Series of data terminals.

9.5 OPERATION

9.5.1 SYSTEM VERIFICATION

Verify the following conditions before applying power to the system:

- Power connected to correct pins on P1 connector
- Jumpers in correct positions (see paragraph 9.4.3 and 9.4.4)
- All application requirements are correctly implemented in the system
- Terminal (if used) connected to correct port on microcomputer board
- Baud rate of terminal (if used) corresponds to that set in user's application program.
- Communications mode is correctly set at terminal and terminal is ON LINE (if used)

9.5.2 POWER-UP/RESET

To power-up and initialize a Target POWER BASIC application, the user must perform the following sequence. This sequence assumes that the application program has been extensively tested on the Host POWER BASIC Interpreter before "configuration", and very likely has also been tested in the final application via the AMPL system. This reasonably ensures that the Target POWER BASIC application to be executed in the final TM990 system is a valid, executable POWER BASIC Interpreter and application. This is necessary since the user will have very little debug capability of the configured application and interpreter at this stage of the cycle. Therefore the user should be reasonably able to assume that the Target POWER BASIC Interpreter/application is at least functional, and should begin execution upon completion of the following power-up sequence.

1) Apply power to board and data terminal (if used)
2) Activate the RESET switch near the corner of the microcomputer board

This should cause the Target POWER BASIC Interpreter to begin execution. The interpreter will first perform internal POWER BASIC

system initialization, followed by execution of the application program.

The user will then need to verify that the application is actually performing the required functions, and should perform a series of tests to check its operation under many conditions and at the limits of its ranges for operation.

If the Target POWER BASIC interpreter/application fails to correctly respond to the preceeding sequence, repeat the sequence, and if it does not respond correctly, proceed to paragraph 9.6 below.

9.6 DEBUG CHECKLIST

If the Target POWER BASIC application does respond correctly, turn the power OFF. Do not turn the power ON again until you are reasonably sure that the problem has been found. The following is a checklist of points to verify.

- Check POWER circuits:
  - Proper power supply voltages and current capacity.
  - Power connections from the power supply to the P1 edge connector. Check pin numbers on P1. Check plug positions at connections. Make sure board is seated in chassis or edge connector socket correctly. Be certain that the edge connector socket (if used) is not upside down.

- Check TERMINAL connection (if used):
  - Proper cable hookup to P2 connector and to terminal. Verify with data in Appendices A and B of the "TM990/101M" or "TM990/100M User's Guide". One of the most common errors is that the terminal cable is not plugged in.
  - Check for power at the terminal. This is another common error - the terminal is not turned ON.
  - Terminal is in ON LINE mode, or equivalent.
  - Terminal is in FULL DUPLEX mode, or equivalent. If the terminal is in HALF DUPLEX mode, it will print everything you type twice, or it may print garbage when you type. Put the terminal in FULL DUPLEX mode.
  - EIA/MD jumper in EIA position.
  - Check BAUD RATE of terminal - it must be 110, 300, 1200, 2400, 4800, 9600, or 19200 BAUD.

- Check microcomputer board jumper plug positions against Table 9-2 or 9-3.

- Check expansion memory board switch positions against Tables 9-6 or 9-7.

- Be sure Target POWER BASIC application EPROM's are in place correctly.

- Verify that all application requirements in the system are correctly implemented.

- Check all socketed parts for correctly inserted pins. Be sure there aren't any bent under or twisted pins. Check pin 1 locations.

If the problem cannot be detected, reapply power and try to feel the components for excessive heat. Be careful as burns may occur is a defective component is found. If the cause of failure cannot be found, it is suggested that the user verify that the EPROM's have the correct contents by using the TXPROM utility of the TX990 system, and have not "lost" any values since they were programmed. If the EPROM's are correct, the user may use AMPL in the system with the final Target POWER BASIC application resident in EPROM to verify various parameters and segments of the program (eg., CRU inputs/outputs, monitoring of terminal values output on the CRU, verification of operation of any "CALLed" assembly language routines, etc.). If the problem still cannot be found, the user should return to the Host POWER BASIC Interpreter and perform additional tests to verify the validity of the application programmed into this application. If the user discovers that the application program functions correctly on the Host POWER BASIC Interpreter, but does not operate correctly in the configured target application, and the user is reasonably confident that the application program is correct to perform the target application, the user should contact the local TI distributor.

APPENDIX A

HOST POWER BASIC INTERPERTER ERROR MESSAGES

POWER BASIC CONFIGURATOR ERROR CODES

TX990 OPERATING SYSTEM ERROR CODES

APPENDIX A-1

HOST POWER BASIC INTERPRETER ERROR MESSAGES

The following error messages may be issued by the HOST POWER BASIC Interpreter:

CODE ERROR MESSAGE
1 = SYNTAX ERROR
2 = UNMATCHED DELIMITER
3 = INVALID LINE NUMBER
4 = ILLEGAL VARIABLE NAME
5 = TOO MANY VARIABLES
6 = ILLEGAL CHARACTER
7 = EXPECTING OPERATOR
8 = ILLEGAL FUNCTION NAME
9 = ILLEGAL FUNCTION ARGUMENT
10 = STORAGE OVERFLOW
11 = STACK OVERFLOW
12 = STACK UNDERFLOW
13 = NO SUCH LINE NUMBER
14 = EXPECTING STRING VARIABLE
15 = INVALID SCREEN COMMAND
16 = EXPECTING DIMENSIONED VARIABLE
17 = SUBSCRIPT OUT OF RANGE
18 = TOO FEW SUBSCRIPTS
19 = TOO MANY SUBSCRIPTS
20 = EXPECTING SIMPLE VARIABLE
21 = DIGITS OUT OF RANGE (0< # of digits <12)
22 = EXPECTING VARIABLE
23 = READ OUT OF DATA
24 = READ TYPE DIFFERS FROM DATA TYPE
25 = SQUARE ROOT OF NEGATIVE NUMBER
26 = LOG OF NON-POSITIVE NUMBER
27 = EXPRESSION TOO COMPLEX
28 = DIVISION BY ZERO
29 = FLOATING POINT OVERFLOW
30 = FIX ERROR
31 = FOR W/O NEXT
32 = NEXT W/O FOR
33 = EXP FUNCTION HAS INVALID ARGUMENT
34 = UNNORMALIZED NUMBER
35 = PARAMETER ERROR
36 = MISSING ASSIGNMENT OPERATOR
37 = ILLEGAL DELIMITER
38 = UNDEFINED FUNCTION
39 = UNDIMENSIONED VARIABLE
40 = UNDEFINED VARIABLE
41 = INVALID END-OF-USER RAM ADDRESS
43 = INVALID BAUD RATE

The following error messages result from the POWER BASIC/TX990 Operating System Interface:

50 = END-OF-FILE OCCURRED
51 = TABLE AREA FULL
52 = INVALID LUNO
53 = INVALID PATHNAME
54 = ZERO LENGTH RECORD
55 = INVALID FILE ACCESS
56 = POSITION ERROR
57 = INCOMPATABLE FILE TYPE

NOTE: The TX990 Operating System error codes as presented in Appendix A-3 may also result during execution of the HOST POWER BASIC Interpreter.

APPENDIX A-2

POWER BASIC CONFIGURATOR ERROR CODES

The Configurator can report most of the error codes presented in Appendix A-1 in addition to the error codes presented below. Appendix A-1 and A-2 errors are reported by the Configurator in the format of "*ERROR XX", where XX is the two character decimal error code corresponding to the error descriptions presented in Appendix A-1 and in the table below.

CODE ERROR MESSAGE

44 = STATEMENT CANNOT BE CONFIGURED - STATEMENT LINE IGNORED
45 = SYNTAX ERROR IN CALL STATEMENT
46 = CALL TABLE FULL - # OF DISTINCT SUBROUTINE CALLS EXCEEDS 16

APPENDIX A-3

TX 990 OPERATING SYSTEM ERROR CODES

The following error codes are reported by the TX990 Operating System. These errors are reported to the user as hexadecimal codes in the format of "FILE I/O ERROR 0XXH", where XX is the two character hexadecimal error code corresponding to the error descriptions presented below.

Code (Hexadecimal)	Description
	DSR ERRORS
01	ILLEGAL LUNO
02	ILLEGAL OPERATION
03	LUNO IS NOT YET OPENED
04	RECORD LOST DUE TO POWER FAILURE
05	ILLEGAL MEMORY ADDRESS
06	TIME OUT, OR ABORT
07	READ CHECK ERROR
11	DEVICE ERROR
12	NO ADDRESS MARK FOUND
15	DATA CHECK ERROR
19	DISKETTE NOT READY
1A	WRITE PROTECT
1B	EQUIPMENT CHECK ERROR
1C	INVALID TRACK OR SECTOR
1D	SEEK ERROR OR ID NOT FOUND
1E	DELETED SECTOR DETECTED
	FILE MANAGEMENT ERRORS
20	LUNO IS IN USE
21	BAD DISC NAME
22	PATHNAME HAS A SYNTAX ERROR
23	ILLEGAL FUR OPCODE
24	BAD PARAMETER IN PRB
25	DISKETTE IS FULL
26	DUPLICATE FILE NAME
27	FILE NAME IS UNDEFINED
28	ILLEGAL LUNO
29	SYSTEM BUFFER AREA FULL
2A	SYSTEM CAN'T GET MEMORY
2B	FILE MANAGEMENT ERROR
2C	CAN'T RELEASE SYSTEM LUNO
2D	FILE IS PROTECTED
2E	ABNORMAL FUR TERMINATION
2F	FILE UTILITY DOES NOT EXIST IN SYSTEM

Code (Hexadecimal)	Description
30	NON-EXISTENT RECORD
3B	INVALID ACCESS PRIVILEGE
3E	FILE CONTROL BLOCK ERROR
3F	FILE DIRECTORY FULL
	TASK LOADER ERROR
60	I/O ERROR, LOAD NOT COMPLETE
61	OBJECT MODULE CONTAINS NONRELOCATABLE OBJECT CODE
62	CHECKSUM ERROR LOAD ABORTED
63	LOADER RAN OUT OF MEMROY
64	TASK 10 IS BUSY
65	IMAGE FILE ERROR
	VDT ERRORS
80	DEVICE NOT AVAILABLE VDT STATION NOT FOUND

Note:

Error Code >FF is a general error code.

APPENDIX B

POWER BASIC

STATEMENT AND COMMAND SUMMARY

EDIT MODE COMMANDS

An advanced editor is contained in POWER BASIC to aid in program writing, editing, and debugging. Note that no editing functions are supported in the "Configured" or Target POWER BASIC Interpreter. The special characters used to perform these editing functions are listed below for the 911VDT, and 913VDT terminal devices. Note that the phrase "(ctrl)" indicates the user holds down the control key while depressing the key corresponding to the character immediately following. Also note that F0, F1, F2 refer to the funtion keys in the top row of the 911 and 913 Video Display Terminals.

911VDT	913VDT	EXAMPLE/EXPLANATION
RETURN ENTER "(ctrl)" M	NEW LINE	(CR) Enter last line typed into program source.
↓(down arrow) "(ctrl)" J	↓(down arrow)	(LF) Enter last line typed into program source and enable auto-line numbering.
"(ctrl)" In F2	F1n	(ctrl)I4 Insert n blanks.
"(ctrl)" Dn F3	F2n	(ctrl)D4 Delete n characters.
"(ctrl)" H ←(left arrow)	←(left arrow)	(ctrl)H Backspace 1 character.
"(ctrl)" F →(right arrow)	→(right arrow)	(ctrl)F Foward space 1 character.
ln "(ctrl)" E ln F1	ln F0	100(ctrl)E Display source line indicated by line number (ln) for editing.
UNLABELED FUNCTION KEY "(ctrl)" 2	RESET	(ESC) Cancel input line or break program execution).
DEL CHAR "(ctrl)" _	DEL CHAR	(DEL) Backspace and delete character.

911VDT	913VDT	EXAMPLE/EXPLANATION
HOME	HOME	HOME Position cursor at column 0, row 0.
UNLABELED KEYPAD KEY	CLEAR	CLEAR Clear screen and position cursor at column 0, row 0.

POWER BASIC commands direct and control system operations. Commands cause immediate computer interaction thereby allowing operator control. Commands may only be entered one per line and may not be entered into a BASIC program. For this reason a "configured" POWER BASIC application may contain no POWER BASIC commands.

SYNTAX	EXAMPLE/EXPLANATION
CONtinue	CONTINUE
	Execution continues from last break.
LIST [exp] -[exp]	LIST 10-100
	Selectively lists the user's POWER BASIC program.
LOAD <$VAR>	LOAD "DSC2:PROCESS/CNT
	Reads a previously recorded POWER BASIC program from specified pathname.
NEW	NEW
	Clear current user program, variables, pointers, and stacks, and prepares for entry of new program.
NEW <address>	NEW 0A000H
	Sets the upper RAM memory bound used by POWER BASIC after auto-sizing at power-up.
NUMBER [start-line] [, inc]	NUMBER 200,20
	Specify starting line number and increment value for auto-line numbering.

SYNTAX	EXAMPLE/EXPLANATION
PURGE <start-line> TO <end-line>	PURGE 200 TO 400 Delete specified range of POWER BASIC statements from user program.
RUN	RUN Begin program execution at the lowest line number.
SAVE <$VAR>	SAVE "CS1" Records a POWER BASIC program to the specified pathname.
SIZE	SIZE Display current program size, allocated variable space, and available memory in bytes.
SOUrce	SOURCE Display number of bytes that will be stored if the current program were SAVEd.
STAck	STACK Display the return line numbers which were pushed on the GOSUB stack.

STATEMENTS

POWER BASIC statements form the basis of all BASIC programs. Statements are typically entered into a program with line numbers and are executed when the RUN command is entered. Statements may also be entered in the keyboard mode without a line number and they will be executed immediately. POWER BASIC statements may occupy only one line; however, numerous statements may appear on each line when delimited by a pair of colons (::). All letters of BASIC statements must be entered in upper case. Not all POWER BASIC statements which execute on the Host POWER BASIC Interpreter may be "configured" into the Target POWER BASIC application. Any explanation preceded by an asterisk (*) indicates that the statement is supported by the Host POWER BASIC Interpreter, but cannot be included in a "configured" (or Target) POWER BASIC application. A "#" indicates that the statement functions differently on the Host system then in a "configured" POWER BASIC Target system. The user should reference the appropriate paragraphs of this manual for the detailed explanation on their operation.

SYNTAX — EXAMPLE/EXPLANATION

ln BAUD <exp1>,<exp2>

BAUD 0,5

Sets the baud rate of the serial I/O port(s) in the "configured" application.

ln BASE <(exp)> BASE (256)

Sets CRU base for subsequent CRU operations.

ln BYE BYE

* Terminates POWER BASIC and returns control to the TX990 operating system.

ln CALL <string-constant>, <subroutine address> [,var1][,var2][,var3][,var4]

CALL "SUB1", ODEOOH,A,(B)

#Transfers to assembly language subroutines. If variable is contained in parenthesis, then address will be passed; otherwise, the value will be passed. Parameters are passed in R4, R5, R6, and R7. Return address is contained in R11.

SYNTAX

EXAMPLE/EXPLANATION

ln DATA {<exp> / <string-constant>} , [{<exp> / <string-constant>}] ....

DATA 1, 4*ANT(1), "HI"

Define internal data block for access by READ statement.

ln DEF FN<x>[(<arg1>[,arg2][,arg3])] =<exp>

DEF FNA(X,Y)=(3*X+Y)/Y

Defines user arithmetic function.

ln DIGITS <exp> DIGITS 7

* Specifies the number of digits to be output.

ln DIM <var (dim[,dim]...)> [,...]

DIM A(10), DOG(5,10,10)

Allocates user variable space for dimensioned or array variables.

ln ELSE <statement(s)>

ELSE GOTO 1000

When most recently executed IF condition is false, all subsequent ELSE statements are executed; otherwise, the ELSE statement line is ignored.

ln END

END

Terminates program execution and returns to keyboard code.

ln EQUATE <sim-var>,<var>[;<sim-var>, var] ....

EQUATE NAM, A(0,5); B5,B(5)

* The specified simple variable is assigned the value of the second variable.

SYNTAX	EXAMPLE/EXPLANATION
ln ERROR <ln>	ERROR 1000 Specifies a subroutine that will be called via an internal GOSUB statement when an error occurs.
ln ESCAPE	ESCAPE * Enables the escape key to interrupt program execution.
ln NOESC	NOESC * (see NOESC statement)
ln FOR <sim-var>=<exp> TO <exp> [STEP <exp>]	FOR I=1 TO 20 STEP 2 The FOR statement is used with the NEXT statement to open and close a program loop. Both identify the same control variable. The FOR statement specifies the control variable and assigns the starting, ending, and optionally stepping values.
ln NEXT <sim-var>	NEXT I (see NEXT statement)
ln GOSUB <ln>	GOSUB 2000 Transfer program execution to an internal BASIC subroutine beginning at the specified line number.
ln POP	POP (see POP statement)

SYNTAX	EXAMPLE/EXPLANATION
ln RETURN	RETURN (see RETURN statement)
ln GOTO <ln>	GOTO 300 Transfer program execution to the specified line number.
ln IF <condition> THEN <statement(s)>	IF I=0 THEN I=J::GOTO 200 Causes conditional execution of the statement(s) following THEN. Statements following THEN execute on TRUE condition.
ln ELSE <statement(s)>	ELSE A=SQR(J)::GOTO 250 (see ELSE statement)
ln IMASK <level>	IMASK 8 # Set interrupt mask of TMS 9900 Microprocessor on target system to specified level.
ln TRAP <level> TO <ln>	TRAP 9 TO 1000 # (see TRAP statement)
ln IRTN	IRTN # Return from BASIC Interrupt service routine.

SYNTAX	EXAMPLE/EXPLANATION
ln INPUT {<num-var> / <string-var>} {, / ;} {<num-var> / <string-var>} .... {;}	INPUT I, $B Places numeric and string values entered from the keyboard into variables in the INPUT list.
ln [LET] <var> = <exp>	LET A=B*4 Evaluates and assigns values to variables or array elements. The LET is optional.
ln NEXT <sim-var>	NEXT I Delimits end of FOR loop. The sim-var must match the FOR control variable.
ln NOESC	NOESC * Disable the ESCape key, to disallow a program break.
ln ON <exp> THEN {GOTO / GOSUB} <ln> [;ln]	ON I THEN GOTO 100,200,300 ON J THEN GOSUB 500,600,700 Case statement used to transfer program execution via a GOTO or GOSUB to the line number specified by the expression.
ln POP	POP Removes from the GOSUB stack the last pushed return address without an execution transfer.

SYNTAX	EXAMPLE/EXPLANATION
ln PRINT <exp> [,exp]...	PRINT A,B,$NAM Print (without formatting) the evaluated expressions to the terminal device.
ln RANDOM <exp>	RANDOM 4*MEM(OFDOOH) Set the seed of the random-number generation to the evaluated expression.
ln READ <num-var> / <string-var> , [<num-var> / <string-var>] ...	READ A,$B,C(0),$D(0) Assigns values from the internal data list to variables or array elements.
ln REM text	REM comment lines for documentation. Inserts comment lines into program.
ln RESTOR [ln]	RESTOR RESTOR 40 RESTOR without a parameter resets pointer to beginning of DATA sequence, while RESTOR with a parameter resets pointer to specified line number.
ln RESTOR # <exp>	RESTOR #I RESTOR with a "# exp " rewinds the logical unit number (LUNO) specified by the expression
ln RETURN	RETURN Return from BASIC subroutine and remove top address from GOSUB stack.

SYNTAX	EXAMPLE/EXPLANATION
ln SPOOL <exp1> TO <exp2>	SPOOL 2 TO A * Directs unit output specified by exp1 to logical unit number (LUNO) specified by exp2.
ln STOP	STOP Terminate program execution and return to keyboard (edit) mode.
ln TIME <exp>,<exp>,<exp>	TIME 11,24,30 # Start the 24-hour time-of-day clock and set the time to the specified expression values.
ln TIME	TIME Output the clock time as HR:MN:SD to the terminal device.
ln TIME <string-var>	TIME $A(0) Stores current clock time into specified string variable.
ln TRAP <level> TO <ln>	TRAP 9 TO 1000 # Assigns interrupt level to BASIC interrupt servicing subroutine.
ln UNIT <exp>	UNIT 3 # Designate the device(s) to receive all printed output.

DISK FILE CONTROL STATEMENTS

The following Host POWER BASIC Statements provide the capability for complete file support on the FS990 system. None of these statements may be "configured" into a POWER BASIC application (target system).

SYNTAX	EXAMPLE/EXPLANATION
ln BDEL {<string variable> / <string constant>}	BDEL "DSC1:BASE/TST" * Deletes the specified diskette file.
ln BCLOSE <numeric variable>	BCLOSE J * Closes specified file (LUNO). (Note that numeric variable is set by the BOPEN statement.)
ln RESET	RESET * Closes all files (LUNO's).
ln COPY <numeric variable> TO <numeric variable>	COPY I to J * Copies contents of one file (LUNO) to another.
ln BDEFR {<string variable> / <string constant>}	BDEFR "DSC2:TEST/REL" * Defines a relative record diskette file.
ln BDEFS {<string variable> / <string constant>}	BDEFS "DSC2:TEST/SEQ" * Defines a sequential diskette file.

SYNTAX	EXAMPLE/EXPLANATION

BOPEN <string variable> .
<string constant> ,<numeric variable>

BOPEN "DSC2:FILE1/SYS",F1

* Opens specified file and assigns LUNO.

ln BINARY 1, <numeric variable>,<exp>

BINARY 1,F,66; 2, A(0)

* Sets BINARY LUNO and specifies the number of bytes to be read or written by subsequent BINARY commands.

ln BINARY 2, <exp> [,exp] ...

BINARY 2, B(1)

* Writes an assigned number of bytes from each expression to the BINARY LUNO. (The BINARY LUNO and number of bytes are assigned by BINARY 1.)

ln BINARY 3, <numeric variable> [,<numeric variable>] ...

BINARY 3 A,B,C

* Enables the user to read an assigned number of bytes into each variable from the BINARY LUNO. (The BINARY LUNO and number of bytes are assigned by BINARY 1.)

ln BINARY 4 ,<exp1>,<exp2>,<exp3>

BINARY 4, 6, 100, 0

* Position within a random file to the specified given byte position.

FUNCTIONS

POWER BASIC provides several predefined mathematical, string, and system functions which simplify program entry and development. Any POWER BASIC function may be used in any statement where a variable may be used. A function is called by using "function name (argument)", where the function name is the three-letter name and the argument maybe any expression or variable. The specified function of the argument replaces the function name in the statement in which it is used. Any explanation preceded by an asterisk (*) indicates that the function is supported by the Host POWER BASIC Interpreter, but cannot be included in a "configured" (or Target) POWER BASIC application.

SYNTAX	EXAMPLE/EXPLANATION
ABS (<exp>)	A=ABS(B) Absolute value of expression
ASC (<string>)	B=ASC($A) (see ASCII Character Conversion Function under STRINGS)
ATN (<exp>)	A=ATN(1) Arctangent of expression. (expression in radians)
BIT (<var>,<exp>)	A=BIT (B,I) Reads bit specified by expression within the specified variable. Returns a 1 if bit is set and a 0 if not set.
BIT (<var>,<exp1>)=<exp2>	BIT (C,6)=1 Modifies bit specified by exp1 in specified variable. Selected bit is set to 1 if assigned value (exp2) is non-zero and to zero if the assigned value (exp2) is zero.

SYNTAX	EXAMPLE/EXPLANATION
COS (<exp>)	A=COS(B) Cosine of expression. (expression in radians)
CRB (<exp>)	A=CRB(-1) Reads CRU bit as selected by the CRU hardware base + exp. Exp is valid over range -127 thru 128.
CRB (<exp1>)=<exp2>	CRB(-4)=0 Set or reset CRU bit as selected by CRU hardware base + exp1. If exp2 is non-zero, the bit will be be set, else reset. Exp1 is valid for -127 thru 128.
CRF (<exp1>)	A=CRF(4) Read n CRU bits as selected by CRU base where exp. evaluates to n. Exp is valid for 0 thru 15. If exp=0, 16 bits will be read.
CRF (<exp1>)=<exp2>	CRF(5)=0FH Output exp1 bits of exp2 to CRU lines as selected by CRU BASE. Exp1 is valid for 0 thru 15. If exp1=0, 16 bits will be output.
EXP (<exp>)	A=EXP(B) Raise the constant e to the power of the expression.
FRA (<exp>)	A = FRA(B) * Return the fractional part of the expression.

SYNTAX	EXAMPLE/EXPLANATION
INP (<exp>)	A=INP(B) Return the signal integer part of the expression.
LEN (<string>)	A=LEN($B(0)) (See String Length Function under STRINGS)
LOG (<exp>)	A=LOG(B) Return natural logarithm of the expression.
MCH (<string1>, <string 2>)	M=MCH($A,$B(0)) (see Character Match Function under STRINGS)
MEM (<exp>)	A=MEM(OFFOOH) A=MEM(65280) Read byte from user memory at address specified by exp.
MEM (<exp1>)= exp2	MEM(OBOH)=OFH MEM(176)=15 Store byte exp2 into user memory at address specified by exp1.
NKY (<exp>)	A=NKY(0) IF NKY(65) THEN PRINT "A" Conditionally samples the keyboard in run-time mode. If exp=0, return the decimal value of last key struck and clear key register. (Zero is returned if no key was struck). If exp 0, compare last struck key with decimal value of exp. If they are the same, a value of 1 is returned and the key register is reset, if not equal then return a 0.

SYNTAX	EXAMPLE/EXPLANATION
RND	A=RND Returns a random number between 0 and 1.
SGN (<exp>)	A=SGN(B) * Returns: 1 if expression is positive 0 if expression is zero -1 if expression is negative
SIN (<exp>)	A=SIN(B) Sine of the expression (exp in radians).
SQR (<exp>)	A=SQR(B) Square root of expression.
SRH (<string1>,<string2>)	S=SRH($A,$B(0)) (See Character Search Function under STRINGS).
SYS (<exp>)	A=SYS(2) Obtain system parameters generated during program execution. The exp values and corresponding system parameters are as follows: SYS(0)=Input control character SYS(1)=Error code number SYS(2)=Error line number
TAN (<exp>)	A=TAN(B) * Tangent of expression (exp in radians).

SYNTAX	EXAMPLE/EXPLANATION
TIC (<exp>)	T1=TIC (0) T2=TIC (T1)

Returns the number of time TICs less the expression value.

STRINGS

Strings variables in POWER BASIC are designated by preceding the variable name with a dollar sign ($). ASCII character strings are stored in contiguous memory byte locations with a null character terminating the string. Hence, simple string variables in POWER BASIC which are 6 bytes in length, can contain up to 5 characters. Dimensioned string variables in POWER BASIC may contain up to the number of elements times 6, less one character. Also, dimensioned string variables may have a byte index following the subscript(s) to indicate a byte position within the specified string. This is indicated by following the subscripts with a semicolon and the byte displacement (e.g., $A(0;5)).

FUNCTION/SYNTAX	EXAMPLE/EXPLANATION
ASCII Character Conversion Function ASC (<string-var>)	ASC($A(0)) Convert first character of string to decimal ASCII Numeric representation.
Assignment <string-var>={<string-variable> / <string-constant>}	$A(0)=$B(0) $A(0)="ABCDE" Store string into string-variable ending string with a null.
Character Match Function MCH (<string1>, <string2>)	A=MCH("HI",$B(0)) Return the number of characters to which the two strings agree.
Character Search Function SRH (<string1>, <string2>)	A=SRH ("BC", "ABCD") Return the position of string1 in string2. Zero is returned if not found.

FUNCTION/SYNTAX	EXAMPLE/EXPLANATION
Concatenate string-var = {<string-variable> / <string-constant>} + <string-var> / <string-constant> [+ ... .]	$A(0)=$A(0)+"END" Concatenate specified string variables or string constants.
Convert to ASCII <string-var>= exp <string-var>= # string>, <exp>	$N(0)=N $N(0)=#"99,990.99",TX Convert exp to ASCII character string ending with a null. "#String" specifies a formatted conversion. (See PRINT OPTIONS.)
Convert to Binary <var1>=<string>, <var2>	N="1234",E N=$A,E Convert string into binary equivalent. Var2 receives the delimiting non-numeric character in the first byte.
Deletion <string-var>=/<exp>	$A(0;3)=/2 Delete exp characters from string var.
Insertion <string-var>=/{<string-variable> / <string-constant>}	$A(0;3)=/"HI" Insert string into specified position in string variable.

FUNCTION/SYNTAX	EXAMPLE/EXPLANATION
String Length Function <num-var>=LEN({<string-variable>}) {<string-constant>}	L=LEN($A(0)) L=LEN "ABC" Return the length of string up to terminating null.
Pick <string-var1>={<string-variable2>}.,<exp> {<string-constant2>}	$I=$A(0;2),3 $J="ABCDE",3 Pick exp number of characters from string2 into string-var1 ending string will null.
Replace <string-var>={<string-variable2>};<exp> {<string-constant2>}	$B(0;2)=$A(0);1 $B(0;2)="...";2 Replace exp number of characters of string-var1 with string2.

INPUT OPTIONS

The following options are available for use with the INPUT statement to provide the POWER BASIC user with enhanced terminal input capability. For additional details on their use, refer to the INPUT statement, Section 5, paragraph 5.8.1.

SYNTAX	EXAMPLE/EXPLANATION
string-var	INPUT $A Prompt with colon and input character data.
,	INPUT A,B Delimit expressions for multiple inputs
;	INPUT ;A INPUT A; Suppress prompting if before variable or suppress CR LF if at end of line.
#exp	INPUT #1"Y or N"$I Specify a maximum of exp characters to be entered.
%exp	INPUT % 4"CODE"C Requires entry of exactly exp number of characters.
? ln	INPUT ?100;A Upon an invalid input or entry of a control character, a GOSUB is performed to the specified line number (ln). SYS(0) will be equal to -1 if there was an invalid input. Otherwise, SYS(0) will equal the decimal equivalent of the control character.

SYNTAX	EXAMPLE/EXPLANATION
"STRING"	INPUT "YES or NO?";$A Prompt with string and then get input. Equivalent to: PRINT "YES or NO?";::INPUT;$A
@(⟨exp1⟩,⟨exp2⟩)	INPUT @ (10,20);A * Move cursor of video display terminal (VDT) to position specified by exp1 and exp2 before performing input variable assignment.
@{⟨string-constant⟩ / ⟨string-variable⟩}	INPUT @"C";A * Perform VDT screen commands as indicated under PRINT options.

OUTPUT OPTIONS

POWER BASIC provides the following options for use with the PRINT statement. They provide powerful print formatting capability for all user output directed to the terminal and/or auxillary device (see UNIT statement). For additional information on these formatting options, refer to Section 5, paragraph 5.8.2.1

SYNTAX	EXAMPLE/EXPLANATION
;	PRINT A;B PRINT A; Delimits expressions or suppress CR LF if at end of line.
,	PRINT A,B Tab to next print field.
TAB (<exp>)	PRINT TAB (50);A Tab to column specified by exp.
string	PRINT "HI";$A(0) Print string or string-var.
# exp	PRINT # 123 Print exp as hexadecimal in free format.
#, exp	PRINT#,50 Print exp as hexadecimal in word format.
#; exp	PRINT #;A Print exp as hexadecimal in byte format.

SYNTAX	EXAMPLE/EXPLANATION
<hex value>	PRINT "<0D><0A>" Direct output of ASCII codes.
#string	PRINT #"99.00"123 Print under specified format where:
	PRINT #"9999"I 9 = digit holder
	PRINT #"000-00-0000"SS 0 = digit holder or force 0
	PRINT #"$$$$,$$$.00"DLR $ = digit holder and floats $
	PRINT #"SSS.0000"4*ATN1 S = digit holder and floats sign
	PRINT #"<<<.00> "I <> = digit holder and float on negative number
	PRINT #"990.99E"N E = sign holder after decimal
	PRINT # "990.99"N . = decimal point specifier

SYNTAX

EXAMPLE/EXPLANATION

```
PRINT #"990.00"N
```

, = suppressed if before significant digit

```
PRINT #"999,990 00"I
```

= translates to decimal point

```
PRINT #"HI=99"I
```

Any other character is printed.

@ (<exp1>,<exp2>)

```
PRINT @ (10,15); "READY"
```

* Move cursor of video display terminal (VDT) to position specified by exp1 and exp2 before performing specified output.

@ {<string-constant> / <string-variable>}

```
PRINT @"C10D2R";N
```

* Perform VDT Screen commands as indicated:

B = beginning of line
C = clear screen
D = down
H = home cursor
L = left
R = right

Any integer preceeding these commands will repeat them specified number of times.

SPECIAL CHARACTER

The following characters have a special meaning when encountered in program statement lines:

CHARACTER	USE
::	Statement separator when entering multiple statements per line.
!	Tail remark indicator used for comments after program statement.
;	Equivalent to "PRINT" statement

ARITHMETIC OPERATIONS

A=B	Assignment
A-B	Subtraction
B+B, $A+$B	Addition or string concatenation
A*B	Multiplication
A/B	Division
A^B	Exponentiation
-A	Unary Minus
+A	Unary Plus

LOGICAL OPERATIONS

The logical operators perform "bit-wise" operations on the operand(s). The operands are converted to 16-bit integer quantities before the operation, and the results of the operation are similarly 16-bit values.

LNOT A	1's complement of integer
A LAND B	Bit wise AND.
A LOR B	Bit wise OR.
A LXOR B	Bit wise exclusive OR.

RELATIONAL OPERATORS

The relational operators are binary operators that operate on two arithmetic expressions. They return values of 1 (TRUE) or 0 (FALSE).

A=B	TRUE if equal, else FALSE.
A==B	TRUE if approximately equal (±1E-7), else FALSE
A<B	TRUE if less than, else FALSE.
A< =B	TRUE if less than or equal, else FALSE.
A>B	TRUE if greater than, else FALSE.
A>=B	TRUE if greater than or equal, else FALSE.
A<>B	TRUE if not equal, else FALSE.

BOOLEAN OPERATORS

The boolean operators are designed to work on the resultant TRUE (1) or FALSE (0) conditions set by the relational operators. However they may also operate on variables within the program, in which case a zero value variable is considered FALSE (0) and a non-zero value variable is considered to be TRUE (1). The boolean operators return value of 1 (TRUE) or 0(FALSE).

NOT A	TRUE if zero, else FALSE
A AND B	TRUE if both non-zero, else FALSE.
A OR B	TRUE if either non-zero, else FALSE.

OPERATOR PRECEDENCE

1. Expressions in parentheses
2. Exponentiation and negation
3. *,/
4. +,
5. <=,<>
6. >=,<
7. =,>
8. ==,LXOR
9. NOT,LNOT
10. AND,LAND
11. OR,LOR
12. (=) ASSIGNMENT

APPENDIX C

POWER BASIC

EXAMPLE PROGRAMS

C.1 SINE WAVE

This sample program demonstrates the use of the TAB function. It will plot a sine wave given input from the user.

```
LIST
5    INPUT "HOW MANY CHARACTERS PER LINE ON YOUR TERMINAL?;M
10    INPUT "MAGNITUDE"A;" PERIOD"B;" #STEPS"C
15    PRINT
20    FOR I=1 TO C
30     PRINT  TAB(INP(M/2)+A*SIN(I/B));"*"
40    NEXT I
50    STOP

RUN
HOW MANY CHARACTERS PER LINE ON YOUR TERMINAL? 80
MAGNITUDE? 38 PERIOD? 3 STEPS? 30
```

```
STOP AT 50
```

```
RUN
HOW MAY CHARACTERS PER LINE ON YOUR TERMINAL ? 80
MAGNITUDE? 30 PERIOD? 4  STEPS? 25
```

```
STOP AT 50
```

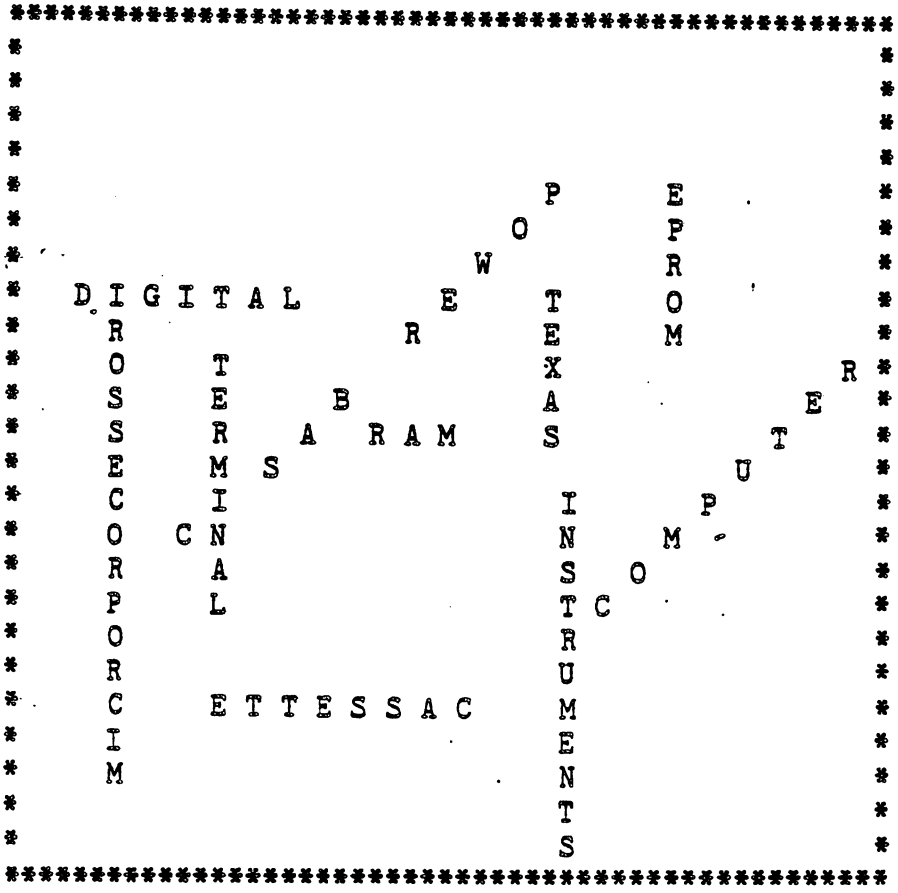
C.2 WORD PUZZLE

This sample program demonstrates character array manipulation. It will hide up a twenty one user-selected words in an array of random letters.

```
100  DIM A(22,5),N(20,7),C(6)
110  $C(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
1000  REM INIT ARRAY
1010  FOR I=0 TO 22:: FOR J=1 TO 23
1020    $A(I,0;J)=" "
1030   NEXT J:: NEXT I
1100  REM ENTER DATA
1110  FOR N=0 TO 20
1120   INPUT $N(N,0)
1130   IF $N(N,0)="" THEN  GOTO 1150
1140  NEXT N
1150  N=N-1 !GET NUMBER OF NAMES
1200  REM FIT INTO ARRAY
1210  FOR I=0 TO N
1220   TRY=0 !NUMBER OF TRYS
1230   TRY=TRY+1
1240   IF TRY-INP(TRY/100)*100=0 THEN PRINT TRY"TH TRY TO FIT "$N(I,0)
1250   IF TRY>500 THEN GOTO 1210  !START AGAIN
1260   R=INP(RND*22):: R1=R
1270   C=INP(RND*22)+1:: C1=C
1280   M=INP(RND*8)+1
1290   IF M=1 THEN RR=-1:: CC=0
1300   IF M=2 THEN RR=-1:: CC=1
1310   IF M=3 THEN RR=0:: CC=1
1320   IF M=4 THEN  RR=1:: CC=1
1330   IF M=5 THEN RR=1:: CC=0
1340   IF M=6 THEN RR=1:: CC=-1
1350   IF M=7 THEN RR=0:: CC=-1
1360   IF M=8 THEN RR=-1:: CC=-1
1370   FOR J=1 TO 19
1380    IF $N(I,0;J)="" THEN GOTO 1500
1400    IF " "=$A(R,0;C),1 THEN GOTO 1430
1420    IF $N(I,0;J)<>$A(R,0;C),1 THEN GOTO 1230  !FIT FAILED, TRY AGAI
1430    R=R+RR:: C=C+CC  !MOVE TO NEXT
1432    IF R>-1 THEN IF R<23 THEN IF C>0 THEN IF C<24 THEN GOTO 1440
1436    IF $N(I,0;J+1)<>"" THEN GOTO 1230
1440   NEXT J
1500   N(I,5)=R1:: N(I,6)=C1:: N(I,7)=M  !SUCCESS!! SAVE POSITION
1510   FOR K=1 TO J-1  !DO ACTUAL MOVE
1520    $A(R1,0;C1)=$N(I,0;K);1:: R1=R1+RR:: C1=C1+CC
1530   NEXT K
1540  NEXT  I
```

```
1550  INPUT "DO YOU WANT TO SEE THE ANSWERS?"%1;$I
1560  IF $I="Y" THEN GOSUB 2000:: GOSUB 3000
1800  REM FILL IN PUZZLE
1810  FOR I=0 TO 22:: FOR J=1 TO 23
1820    $L=$A(I,0;J),1
1830    IF " "=$A(I,0;J),1 THEN $A(I,0;J)=$C(0;INP(RND*26)+1);1
1840   NEXT J:: NEXT I
1850  GOSUB 2000
1860  STOP
2000  REM PRINT ARRAY
2005  PRINT :: PRINT
2010  FOR I=1 TO 49:: PRINT "*";:: NEXT I:: PRINT
2020  FOR I=0 TO 22:: PRINT "* ";:: FOR J=1 TO 23
2030    $L=$A(I,0;J),1:: PRINT $L" ";
2040   NEXT J:: PRINT "*":: NEXT I
2050  FOR I=1 TO 49:: PRINT "*";:: NEXT I:: PRINT
2060  RETURN
3000  FOR I=0 TO N
3010   PRINT $N(I,0) TAB 20;N(I,5)+1;N(I,6);N(I,7)
3020  NEXT I
3030  RETURN

RUN
:  POWER BASIC
:  TEXAS INSTRUMENTS
:  COMPUTER
:  MICROPROCESSOR
:  TERMINAL
:  CASSETTE
:  DIGITAL
:  RAM
:  EPROM
:
DO YOU WANT TO SEE THE ANSWERS?Y
```



POWER BASIC	4 14 6
TEXAS INSTRUMENTS	7 15 5
COMPUTER	16 16 2
MICROPROCESSOR	21 3 1
TERMINAL	9 6 5
CASSETTE	19 13 7
DIGITAL	7 2 3
RAM	11 10 3
EPROM	4 18 5

```
**************************************************
* S A U B R H N H F L S W R G K T R V U D C L I *
* L A Q V H W F E R D F S R U Y V V X Y H S V A *
* C G D E M R T U Y K U I E B N Q E O Q S E D Z *
* X R E R T L I A B J O R V T P A M E K C D O H *
* B P H I S H F N G J G F D O I M X P D U H K F *
* R G U P E H I U S L P O W Y C J V R U V Q W L *
* O D I G I T A L U D I E J J T J B O Q C P Q Y *
* C Z R E D P C C J F R P D Y E R M M D N C A I *
* Y S O R C T I D E D A C N J X Q R Y Y B E X R *
* H I S D E E C W B I S H H V A T A K Y U S E J *
* F O S A M R K A B R A M X K S O M N A B T I I *
* J J E I X M S W H N G C X B N X M M D U L P N *
* T A C A C I H G U O Y B G H I D M X P B N X O *
* W A O Y C N G Y V M I S I O N H A M G P Y T M *
* E I R E E A E T V T R T I Y S A O D U P X Y J *
* Y C P M N L K T H B P G E Y T C L D E K F I O *
* X F O X X F E F A G A X Y I R X U Z X G A Q S *
* S Y R S S G E I E B O S O M U E K P F L G H F *
* Y G C X Q E T T E S S A C O M T V X J G X R S *
* P A I X G U M S V D T O O F E O C T T X G C W *
* Z F M J W I J T F Q L O T X N I K O K E E X O *
* L Z H C T D P B C H V R P G T G K R F M Z A L *
* G J O Q Q W L P Y W D L S F S G O T U T N B X *
**************************************************
```

STOP AT 1860

APPENDIX D

FLOATING POINT PACKAGE

D.1 INTRODUCTION

This appendix contains the documentation of the floating point package. The package may be accessed by an assembly language program through the use of XOPs.

The POWER BASIC Floating Point package is a single accumulator Floating Point Processor. It includes the common operations of addition, subtraction, multiplication and division. Also provided are utilities to load, store, scale, normalize, clear, float and negate.

D.2 SYNTAX

XOP SYNTAX:

```
LABEL  ß..XOP ß.. GA,OP ß.. comment
```

For clarity in explanation, the XOPS will be defined as follows:

```
DXOP  LOADF,0
DXOP  STORE,1
DXOP  FADD,2
DXOP  FSUB,3
DXOP  FMUL,4
DXOP  FDIV,5
DXOP  SCALE,6
DXOP  NORMAL,7
DXOP  CLEAR,8
DXOP  NEGATE,9
DXOP  FLOAT,10
```

D.3 FLOATING POINT FORMAT AND ACCURACY

Detailed information on the format and accuracy of floating point numbers may be found in Section 3.7.7.

D.4 Paragraphs D.4.1 through D.4.7 describe the utilities provided by the floating point package.

D.4.1 LOAD

XOP 0 (LOAD) will load FPAC (Floating Point Accumulator) with the 6 byte number addressed by the operand.

Example:

```
LOADF  @FP1  or XOP @FP1,0
.
.
.
FP1  DATA  >4110,>0000,>0000
```

Will load FPAC with the contents of FP1.

D.4.2 STORE

XOP 1 (STORE) will store FPAC at the 6 byte location addressed by the operand.

Example:

```
        STORE  @FP2   or    XOP  @FP2,1
        .
        .
        .
    FP2 BSS   6
```

Will transfer the contents of FPAC to FP2.

D.4.3 SCALE

XOP 6 (SCALE) will adjust the exponent of FPAC to the value of the operand.

Example:

```
        SCALE  @C4A  or XOP  @C4A,6
        .
        .
        .
    C4A  DATA   >4A00
```

Will adjust FPAC so the exponent becomes >4A.

The SCALE XOP may also be used to convert a floating point value in FPAC to an integer value in FPAC. Note the user should verify that the exponent of the floating point value is not larger than hex 44, otherwise the floating point number is to large and cannot be represented as an integer. The integer format of POWER BASIC is presented in Section 3, Paragrph 3.7.7.

The SCALE XOP with an operand value of hex 4600 will convert the value in FPAC to an integer number as follows:

```
          SCALE  @ C4600     or   XOP  @ C4600, 6
          .
          .
          .
          .
          .
C4600     DATA >4600
```

D.4.4 NORMALIZE

XOP 7 (NORMALIZE) will adjust FPAC such that the first hex digit of the fraction is non-zero.

Example:

```
          NORMAL  0  or  XOP  0,7
```

Will normalize FPAC.

Note that the operand has no significance.

D.4.5 CLEAR

XOP 8 (CLEAR) will zero FPAC.

Example:

```
          CLEAR  0   or XOP  0,8
```

Will zero FPAC.

Note that the operand has no significance.

D.4.6 NEGATE

XOP 9 (NEGATE) will negate FPAC by changing the first bit. If FPAC is zero, it will remain zero.

Example:

```
          NEGATE  0  or XOP 0.9
```

Will Negate FPAC.

Note that the operand has no significance.

D.4.7 FLOAT

XOP 10 (FLOAT) will float the second word of FPAC into a floating point number. This word is a 16 bit 2's compliment integer.

Example:

```
        CLR    R0
        LI     R1,100
        CLR    R2
        LOADF  R0          OR XOP 0,0
        FLOAT  0           OR XOP 0,10
        STORE  @FP100      OR XOP @FP100,1
        .
        .
FP100   BSS    6
```

Will load FPAC with a decimal 100, convert it to floating point and store it in FP100 .

Note that the operand has no significance.

D.5 Paragraphs D.5.1 through D.5.5 describe the mathematical operators of the floating point package. All operators are required to be normalized and the result will be normalized and returned in the floating point accumulator (FPAC). If the result is zero, it is returned as a true zero (i.e. all zeros as opposed to a floating point zero, >4000, >0000, >0000).

D.5.1 XOP 2 (ADDITION) will add the 6 byte number addressed by the operand to the FPAC and place the results in FPAC.

Example:

```
        FADD  @C10          OR XOP  @C10,2
        .
        .
        .
C10     DATA  >41A0, >0000, >0000
```

Will add the contents of C10 to FPAC and place the results in FPAC.

D.5.2 SUBTRACTION

XOP 3 (SUBTRACTION) will subtract 6 byte number addressed by the operand from the FPAC and place the results in FPAC.

Example:

```
        FSUB    @C10            OR   XOP   @C10,3
          .
          .
          .
C10     DATA    >41A0, >0000, >0000
```

Will subrtract the contents of C10 from FPAC and place the results in FPAC.

D.5.3 MULTIPLICATION

XOP 4 (MULTIPLICATION) will multiply FPAC by the 6 byte number addressed by the operand and place the result in FPAC.

Example:

```
        FMUL    @C10            OR   XP    @C10,4
          .
          .
          .
C10     DATA    >41A0, >0000, >0000
```

Will multiply the contents of FPAC by the contents of C10 and place the results in FPAC.

D.5.4 DIVISION

XOP 5 (DIVISION) will divide FPAC by the 6 byte number addressed by the operand and place the result in FPAC.

Example:

```
        FDIV    @C10            OR   XOP   @C10,5
          .
          .
          .
C10     DATA    >41A0,  >0000, >0000
```

Will divide the contents of FPAC by the contents of C10 and place the results in FPAC.

D.5.5 EXAMPLE

For example, the equation

```
A=B+C*D
```

Would become:

```
        LOADF   @C      OR      XOP   @C,0
        FMUL    @D      OR      XOP   @D,4
        FADD    @B      OR      XOP   @B,2
        STORE   @A      OR      XOP   @A,1
        .
        .
        .
A       BSS       6
B       DATA    >4110, >0000, >0000
C       DATA    >4118, >0000, >0000
D       DATA    >4118, >0000, >0000
```

APPENDIX E

CONFIGURATOR EXAMPLES

APPENDIX E

CONFIGURATOR EXAMPLES

The following examples demonstrate the configuration process for two simple example programs. The complete execution sequence will be presented as well as the listings produced by the Configurator and the Link Editor. The configuration process will culminate in execution of the configured example program.

E.1 CONFIGURATION OF SINE WAVE EXAMPLE PROGRAM

The sine wave program presented below will plot a sine wave given input from the user. This program will be developed and thoroughly tested on the Host POWER BASIC Interpreter before continuing to the configuration process presented below. The listing of the program is:

```
LIST
5  REM TEST 4
7  BAUD 0,4.
10  INPUT "HOW MANY CHARACTERS PER LINE ON YOUR TERMINAL?";M
20  INPUT "MAGNITUDE "A;" PERIOD "B;" # STEPS "C
30  PRINT
40  FOR I=1 TO C
50   PRINT  TAB (INP(M/2)+A*SIN(I/B));"*"
60  NEXT I
70 STOP
```

The user will then save the program onto a diskette file as follows:

```
BDEFS "DSC2:TEST4/SRC"
SAVE "DSC2:TEST4/SRC"
```

The diskette files to receive the Configurator outputs must also be defined as follows:

```
BDEFS "DSC2:TEST4/LNK"
BDEFS "DSC2:TEST4/OBJ"
```

The user will then terminate the Host POWER BASIC Interpreter (via the BYE statement) and return to the TX990 Operating System.

Next the Configurator will be executed. For details on the loading and execution of the Configurator, refer to Section 2. The user will respond to the prompts of the Configurator as follows:

will respond to the prompts of the Configurator as follows:

```
POWER BASIC CONFIGURATOR REV C.1.4

  APPLICATION SRCE= DSC2:TEST4/SRC
      LINK CONTROL= DSC2:TEST4/LNK
       OBJECT FILE= DSC2:TEST4/OBJ
         LIST FILE= LP
```

The Configurator will begin executing on the application program and produce the link control file, object file, and the listing output. When the configuration process is complete, the APPLICATION SRCE= prompt will again be displayed. The user will respond with the ESCAPE key (RESET on the VDT913 or unlabeled function key on the VDT911) to terminate Configurator execution.

The listing output produced by the Configurator is shown in Figure E-1. The user should refer to Section 8, paragraph 8.4.3.3 for details on the Configurator list file output.

The Link Editor will then be loaded and executed using the TXDS Control Program. The user responses to these prompts are as follows:

```
TXDS  936215  *A  293/78     11:25

PROGRAM:  DSC:TXSLNK/SYS
  INPUT:  DSC2:TEST4/LNK
 OUTPUT:  DSC2:TEST4/SYS,LP
OPTIONS:  M15000
```

The Link Editor will be called into execution, and will produce the final linked Target POWER BASIC Interpreter/Application.

The listing output produced by the Link Editor is shown in Figure E-2.

The user may then test the final object module (DSC2:TEST4/SYS) by using AMPL to verify the operation of the Target POWER BASIC Interpreter/Application before performing the final step of programming the object file into EPROM.

```
POWER BASIC CONFIGURATOR REV C.1.4

                    MODULE SUMMARY
                    MODULES USED

NAME      PRIMARY REF   SECONDARY REF      NAME      PRIMARY REF   SECONDAR REF
----      -----------   -------------      ----      -----------   ------------
BAUDC           1             6          CVBDC           0             2
CVDB            0             2          CVGC            0             4
FOR             2             0          FUNCC           1             0
GOSUB           0             2          INPUT           2             0
IPCOM           0             4          JMP             0             4
MOVE            0             4          POLY            0             1
PRINTC          2             0          PUTB            0             2
SINF            1             0

POWER BASIC CONFIGURATOR REV C.1.4

                    STATEMENTS USED

NAME      # OF REFS     MODULES
----      ----------    -------
FOR           1         FOR
NEXT          1         FOR
PRINT         2         PRINTC
INPUT         2         INPUT
BAUD          1         BAUDC

POWER BASIC CONFIGURATOR REV C.1.4

APPLICATION SOURCE=DSC2:TEST4/SRC
       LINK CONTROL=DSC2:TEST4/LNK
      OBJECT FILE=DSC2:TEST4/OBJ

( EDIT ERRORS, IF ANY, LISTED HERE )

NUMBER OF BYTES OF PSEUDO SOURCE= >00CC

POWER BASIC CONFIGURATOR REV C.1.4

                    FUNCTIONS USED

NAME      # OF REFS     MODULES
----      ----------    -------
INP           1         FUNCC
SIN           1         SINF
```

FIGURE E-1. CONFIGURATOR LIST FILE OUTPUT

```
 TXSLNK        2.3.1  78.244        01/00/00  00:07:04                    PAGE    1
COMMAND LIST

NOSYMT
NOAUTO
LIBRARY :CBASIC/LIB
TASK CBASIC
INCLUDE :STARTC/OBJ
INCLUDE (BAUDC)
INCLUDE (CVBDC)
INCLUDE (CVDB)
INCLUDE (CVGC)
INCLUDE (FOR)
INCLUDE (FUNCC)
INCLUDE (GOSUB)
INCLUDE (INPUT)
INCLUDE (IPCOM)
INCLUDE (JMP)
INCLUDE (MOVE)
INCLUDE (POLY)
INCLUDE (PRINTC)
INCLUDE (PUTB)
INCLUDE (SINF)
INCLUDE (ZZENDC)
INCLUDE DSC2:TEST4/OBJ
END


SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = ABSF      MODULES = 7, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = BAUD      MODULES = 2, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = COSF      MODULES = 16, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = CVHD      MODULES = 3, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = CVBI      MODULES = 3, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = FORY      MODULES = 6, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = GOS1      MODULES = 8, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = GOSB1     MODULES = 8, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = GOSY      MODULES = 8, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = GOTY      MODULES = 8, 17

SYMBOL MULTIPLY DEFINED ********************************************************
SYMBOL = INPF      MODULES = 7, 17

SYMBOL MULTIPLY DEFINED ********************************************************
```

FIGURE E-2. LINK EDITOR LIST FILE OUTPUT (SHEET 1 of 4)

```
.TXSLNK          2.3.1   78.244          01/00/00   00:19:52
COMMAND LIST

SYMBOL = INPY       MODULES = 9, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = NKYF       MODULES = 2, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = NXIY       MODULES = 6, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = POPY       MODULES = 8, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = PRTY       MODULES = 14, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = RTNY       MODULES = 8, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = SINF       MODULES = 16, 17

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = SYSF       MODULES = 7, 17


 TXSLNK          2.3.1   78.244          01/00/00   00:19:52
LINK MAP

CONTROL FILE = DSC2:TEST4/LNK

LINKED OUTPUT FILE = DSC2:TEST4/SYS

LIST FILE = LP

OUTPUT FORMAT = ASCII

LIBRARIES

NO    ORGANIZATION    PATHNAME

1     SEQUENTIAL      :CBASIC/LIB
```

FIGURE E-2. LINK EDITOR LIST FILE OUTPUT (SHEET 2 OF 4)

PHASE 0, CBASIC ORIGIN = 0000 LENGTH = 1930

MODULE	NO	ORIGIN	LENGTH	TYPE	DATE	TIME	CREATOR
STARTC	1	0000	0B04	INCLUDE	01/02/79	14:08:03	SDSLNK
BAUDC	2	0B04	0138	INCLUDE,1	01/03/79	09:51:17	SDSLNK
CVBDC	3	0C3C	034E	INCLUDE,1	01/03/79	09:52:32	SDSLNK
CVDB	4	0F8A	01B2	INCLUDE,1	01/03/79	09:52:51	SDSLNK
CVGC	5	113C	005E	INCLUDE,1	01/03/79	09:53:06	SDSLNK
FOR	6	119A	01B2	INCLUDE,1	01/03/79	09:54:12	SDSLNK
FUNCC	7	131C	0056	INCLUDE,1	01/03/79	09:54:31	SDSLNK
GOSUB	8	1372	0040	INCLUDE,1	01/03/79	09:55:08	SDSLNK
INPUT	9	1412	0104	INCLUDE,1	01/03/79	09:55:39	SDSLNK
IPCOM	10	1516	003A	INCLUDE,1	01/03/79	09:56:08	SDSLNK
JMP	11	1550	001C	INCLUDE,1	01/03/79	09:56:21	SDSLNK
MOVE	12	156C	0060	INCLUDE,1	01/03/79	09:57:27	SDSLNK
POLY	13	15CC	006B	INCLUDE,1	01/03/79	09:57:51	SDSLNK
PRINTC	14	1634	0134	INCLUDE,1	01/03/79	09:58:21	SDSLNK
PUTB	15	1768	004E	INCLUDE,1	01/03/79	09:58:42	SDSLNK
SINF	16	17B6	00A0	INCLUDE,1	01/03/79	09:59:50	SDSLNK
ZZENDC	17	1856	000E	INCLUDE,1	01/03/79	10:01:02	SDSLNK
ROOT	18	1864	00CC	INCLUDE			CONFIG

D E F I N I T I O N S

NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO
ABSF	1356	7	ADDF	09B4	1	ANDF	1856	17	ASCF	1856	17
ASR0	003C*	1	ASR1	003E*	1	ATNF	1856	17	B01	096A	1
B05	0965	1	B20	01FE	1	B3F	01FF	1	BASY	1856	17
BAUD	0B0B	2	*BCRU	0034*	1	BITF	1856	17	BITY	1856	17
BMVE	156C	12	BUS	000C*	1	*C1	0960	1	C4	0962	1
C4A00	01FC	1	*C6	096A	1	CCNT	002A*	1	*CKEX	0586	1
*CLKADR	FE6C*	1	CLKI	185A	17	*CLKT01	FE72*	1	*CLKT02	FE74*	1
*CLKWS	FE6A*	1	CLLY	1856	17	COSF	17B6	16	CRBF	1856	17
CRBY	1856	17	CRFF	1856	17	CRFY	1856	17	CVBD	0C4A	3
*CVBF	0E4E	3	CVBI	0C40	3	CVC10	1164	5	CVCH	FEEC*	1
*CVDB20	10C4	4	CVDIFZ	0F8E	4	*CVDIZ	0F8A	4	CVGCN	113C	5
CVGCN1	1144	5	CVHD	FEF2*	1	CVHD01	FFF3*	1	CVHD15	FF01*	1
*DATXB	0201	1	*DDM	0022*	1	DFFY	1856	17	DIMY	1856	17
*DLC	0020*	1	DLIM	001E*	1	DS	FF76*	1	DS1	FE7C*	1
*DS2	FF8C*	1	*DS3	FFC2*	1	*EBP	FEF2*	1	*EFLG	002A*	1
ELNM	002E*	1	*ELSF	0030*	1	*ELSY	1856	17	ENUM	002C*	1
ERRY	1856	17	EUS	0002*	1	*EVAL	073C	1	*EVALS2	0734	1
EVARZ	0582	1	EVERZ	05CC	1	*EVFX	05E4	1	*EVOP3A	087E	1
EVSDZ	0574	1	EVSFR	0714	1	*EVSKB	FEF2*	1	*EVSKE	FF94*	1
EXPF	1856	17	*FAD	0246	1	FADDI	051E	1	*FCLR	044E	1
*FDD	0482	1	FFLG	0032*	1	FIX	0A82	1	*FLDD	0234	1
*FLOAT	0300	1	*FMD	0362	1	*FNEG	02F6	1	*FNRM	0328	1
FNS	0004*	1	*FNSZ	0004*	1	*FOR2	11DA	6	FORY	119A	6
FPAC	FF9C*	1	FPAC2	FF9E*	1	FPAC4	FFA0*	1	*FPWP	FF9C*	1
*FSCL	0546	1	*FSD	023C	1	*FSRD	022C	1	FSUBI	0532	1
FUNBK	15CC	13	FUNFX	15F8	13	*FUZZ	3800*	1	*GETC	0BF8	2
GETCR	0BEC	2	GOS1	138E	8	*GOS2	1302	8	GOSB1	1372	8

FIGURE E-2. LINK EDIT FILE OUTPUT (SHEET 3 OF 4)

```
 TXSLNK        2.3.1  78.244        01/00/00  00:07:04                    PAGE    5
 NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO

*GOSON    1382  8      GOSY     1380  8      GOTY     1382  8      GSC      0016* 1
*GSIM     12F6  6      GSS      0006* 1     *GSSZ     0014* 1      HFLG     0034* 1
*HOUT     172C  14      IFLG     0036* 1      IFY      1856  17     IMKY     1856  17
 INPEND   152C  10     INPF     133A  7     INPY     1412  9     *INTFLG   FE56* 1
 INTIN    185A  17    *INTSB    FE36* 1     *INTSP    FE58* 1     *INTWP2   FE48* 1
*IOB      0000* 1      IRTY     1856  17    *ISTCK    FE36* 1     *ITAB     FE16* 1
*ITAB2    FE1A* 1     *JMPR0    155A  11     JMPR0A   1550  11     LANDF    1856  17
 LFC      0024* 1      LENF     1856  17     LETY     1856  17    *LINE     0160  1
 LINE0    016E  1      LINE2    017A  1      LINE5    0160  1      LNOTF    1856  17
 LNSZ     0084* 1      LOGF     1856  17     LORF     1856  17     LXORF    1856  17
 MCHF     1856  17     MEMF     1856  17     MEMY     1856  17     MODE     0040* 1
*NIC      0064* 1      NKYF     0C0A  2      NLIN     015A  1      NLIN0    0156  1
 NOTF     1856  17    *NRV      0004* 1      NVD      0012* 1      NVS      0014* 1
 NXIX8    0200  1      NXTY     1294  6      ONY      1856  17     ORF      1856  17
 OUTL1    1516  10    *PFIX     0ADE  1      PLC      001C* 1      PLF      0038* 1
 PLYX     161E  13     PLYXX    1612  13     POPY     1400  8      POWF     1856  17
 PRTEND   1524  10     PRTY     1634  14     PUTB     1768  15    *RANDS    FE6A* 1
 RANDZ    1860  17     RANY     1856  17     RDDY     1856  17    *REST     0134  1
 RNWY     1856  17     ROOT     1864  18     RTNY     13DE  8      SFSN     0AF0  1
 SINF     17CA  16    *SLN      0026* 1      SLT      000E* 1      SQRF     1856  17
*SQRI     0004* 1      SRHF     1856  17    *SSP      FF76* 1      SUBF     090C  1
 SYSF     131C  7     *TEMP     FF94* 1     *TEMP2    FF96* 1     *TEMP4    FF9A* 1
*TEMP6    FF9A* 1      TICF     1856  17     TIMY     1856  17     TRPY     1856  17
 TY0      001A* 1      TYP0     0F6E  2     *TYP11    0F6C  2      TYPB     0880  2
*TYPBE    0B7E  2      TYPC     0B78  2     *UCCNT    153C  10     UFT      0008* 1
*UMSK     0003* 1      UNIT     001A* 1      UNTY     1856  17     VDT      000A* 1
 VNT      0010* 1     *WPR1     FE8A* 1     *WPR103   FF90* 1     **PR10   FF92* 1
*WPR108   FF9A* 1      WPR2     FFDC* 1


19 WARNINGS

**** LINKING COMPLETED
```

FIGURE E-2. LINK EDITOR LIST FILE OUTPUT (SHEET 4 OF 4)

The configuration process is now complete. When the Target POWER BASIC Interpreter/Application is executed in the TM990 target system, the output pictured below will result.

```
HOW MANY CHARACTERS PER LINE ON YOUR TERMINAL?80
MAGNITUDE ? 35 PERIOD ? 5 # STEPS ? 38
```

E.2 CONFIGURATION OF NUMERIC ENTRY EXAMPLE PROGRAM

This example program will accept numeric user responses and display their square and square root. This program will also be developed and thoroughly tested on the Host POWER BASIC Interpreter before continuing to the configuration process presented below. The listing of the program is:

```
LIST
5  REM TEST1
7  BAUD 0,4
10  DIM A(4)
20  $A(0)="THE NUMBER IS"
30  INPUT "INPUT NUMBER",N
40  IF N-INP(N)<>0 THEN PRINT $A(0);N:: GOTO 60
50  GOSUB 100  !  EVEN OR ODD INTEGER
60  PRINT ", ITS SQUARE IS";N*N;", AND ITS SQUARE ROOT IS";
70  IF N<0 THEN PRINT " UNDEFINED.":: GOTO 30
80  PRINT SQR(N);"."
90  GOTO 30
100  IF INP(N/2)*2=N THEN PRINT $A(0);" EVEN";:: RETURN
110  PRINT $A(0);" ODD";
120  RETURN
```

The user must then save the program onto a diskette file as follows:

```
BDEFS "DSC2:TEST1/SRC"
SAVE "DSC2:TEST1/SRC"
```

The diskette files to receive the Configurator outputs must also be defined as follows:

```
BDEFS "DSC2:TEST1/LNK"
BDEFS "DSC2:TEST1/OBJ"
```

The user will then terminate the Host POWER BASIC Interpreter (via the BYE statement) and return to the TX990 Operating System.

Next the Configurator will be executed. For details on the loading and execution of the Configurator, refer to Section 2. The user will respond to the prompts of the Configurator as follows:

```
POWER BASIC CONFIGURATOR REV C.1.4

APPLICATION SOURCE= DSC2:TEST1/SRC
     LINK CONTROL= DSC2:TEST1/LNK
      OBJECT FILE= DSC2:TEST1/OBJ
        LIST FILE= LP
```

The Configurator will begin executing on the application program and produce the link control file, object file, and the listing output. When the configuration process is complete, the APPLICATION SOURCE= prompt will again be displayed. The user will respond with the ESCAPE key (RESET on the VDT913 or the unlabeled function key on the VDT911) to terminate Configurator execution.

The listing output produced by the Configurator is shown in Figure E-3. The user should refer to Section 8, paragraph 8.4.3.3 for details on the Configurator list file output.

The Link Editor will then be loaded and executed using the TXDS Control Program as follows:

```
TXDS  936215    *A   293/78    11:45

PROGRAM:  DSC:TXSLNK/SYS
  INPUT:  DSC2:TEST1/OBJ
 OUTPUT:  DSC2:TEST1/SYS,LP
OPTIONS:  M15000
```

The Link Editor will be called into execution, and will produce the final linked Target POWER BASIC Interpreter/Application.

The listing output produced by the Link Editor is shown in Figure E-4.

The user may then test the final object module (DSC2:TEST1/SYS) by using AMPL to verify the operation of the Target POWER BASIC Interpreter/Application before performing the final step of programming the object file into EPROM.

The configuration process is now complete. When the Target POWER BASIC Interpreter/Application is executed in the TM990 Target system, the output shown below will result.

```
PUT NUMBER? 45
E NUMBER IS ODD,  ITS SQUARE IS 2025, AND ITS SQUARE ROOT IS 6.7082039.
PUT NUMBER? 87
E NUMBER IS ODD,  ITS SQUARE IS 7569, AND ITS SQUARE ROOT IS 9.3273791.
PUT NUMBER? -2
E NUMBER IS EVEN,  ITS SQUARE IS 4, AND ITS SQUARE ROOT IS UNDEFINED.
PUT NUMBER? 0
E NUMBER IS EVEN,  ITS SQUARE IS 0, AND ITS SQUARE ROOT IS 0.
PUT NUMBER? 256
E NUMBER IS EVEN,  ITS SQUARE IS 65536, AND ITS SQUARE ROOT IS 16.
PUT NUMBER? 8000
E NUMBER IS EVEN,  ITS SQUARE IS 64000000, AND ITS SQUARE ROOT IS 89.442719
PUT NUMBER? FF?? 2
E NUMBER IS EVEN,  ITS SQUARE IS 4, AND ITS SQUARE ROOT IS 1.4142136.
PUT NUMBER? 02H
E NUMBER IS EVEN,  ITS SQUARE IS 4, AND ITS SQUARE ROOT IS 1.4142136.
PUT NUMBER? 0FH
E NUMBER IS ODD,  ITS SQUARE IS 225, AND ITS SQUARE ROOT IS 3.8729833.
PUT NUMBER? 15
E NUMBER IS ODD,  ITS SQUARE IS 225, AND ITS SQUARE ROOT IS 3.8729833.
PUT NUMBER? .5
E NUMBER IS 0.5,  ITS SQUARE IS 0.25, AND ITS SQUARE ROOT IS 0.70710678.
```

```
POWER BASIC CONFIGURATOR REV C.1.4

APPLICATION SOURCE=DSC2:TEST1/SRC
       LINK CONTROL=DSC2:TEST1/LNK
        OBJECT FILE=DSC2:TEST1/OBJ

( EDIT ERRORS, IF ANY, LISTED HERE )

NUMBER OF BYTES OF PSEUDO SOURCE= >0128
POWER BASIC CONFIGURATOR REV C.1.4
```

MODULE SUMMARY
MODULES USED

NAME	PRIMARY REF	SECONDARY REF	NAME	PRIMARY REF	SECONDARY REF
BAUDC	1	8	CVBIDC	0	6
CVIDB	0	2	CVGC	0	8
DIM	1	0	FUNCC	2	0
GOSUB	6	1	IF	3	0
INPUT	1	0	IPCOM	0	7
JMP	0	7	LET	1	0
MOVE	0	7	POLY	0	1
PRINTC	6	0	PUTB	0	1
SQRF	1	0			

```
POWER BASIC CONFIGURATOR REV C.1.4
```

STATEMENTS USED

NAME	# OF REFS	MODULES
GOTO	3	GOSUB
GOSUB	1	GOSUB
(LET)	1	LET
PRINT	6	PRINTC
INPUT	1	INPUT
RETURN	2	GOSUB
BAUD	1	BAUDC
DIM	1	DIM
IF	3	IF

```
POWER BASIC CONFIGURATOR REV C.1.4
```

FUNCTIONS USED

NAME	# OF REFS	MODULES
INP	2	FUNCC
SQR	1	SQRF

FIGURE E-3. CONFIGURATOR LIST FILE OUTPUT

```
 TXSLNK        2.3.1  78.244        01/00/00  00:01:11                PAGE   1
COMMAND LIST

NOSYMT
NOAUTO
LIBRARY :CBASIC/LIB
TASK CBASIC
INCLUDE :STARTC/OBJ
INCLUDE (BAUDC)
INCLUDE (CVBDC)
INCLUDE (CVDB)
INCLUDE (CVGC)
INCLUDE (DIM)
INCLUDE (FUNCC)
INCLUDE (GOSUB)
INCLUDE (IF)
INCLUDE (INPUT)
INCLUDE (IPCOM)
INCLUDE (JMP)
INCLUDE (LET)
INCLUDE (MOVE)
INCLUDE (POLY)
INCLUDE (PRINTC)
INCLUDE (PUTB)
INCLUDE (SQRF)
INCLUDE (ZZENDC)
INCLUDE DSC2:TEST1/OBJ
END


SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = ABSF      MODULES = 7, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = BAUD      MODULES = 2, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = CVBD      MODULES = 3, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = CVBI      MODULES = 3, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = DIMY      MODULES = 6, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = GOSI      MODULES = 8, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = GOSBI     MODULES = 8, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = GOSY      MODULES = 8, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = GOTY      MODULES = 8, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = IFY       MODULES = 9, 19

SYMBOL MULTIPLY DEFINED ********************************************
SYMBOL = INPF      MODULES = 7, 19
```

FIGURE E-4. LINK EDITOR LIST FILE OUTPUT (SHEET 1 OF 4)

```
 .TXSLNK          2.3.1   78.244          01/00/00   00:02:18
COMMAND LIST


SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = INPY       MODULES = 10, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = LETY       MODULES = 13, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = NKYF       MODULES = 2, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = POPY       MODULES = 8, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = PRTY       MODULES = 16, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = RTNY       MODULES = 8, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = SQRF       MODULES = 18, 19

SYMBOL MULTIPLY DEFINED *W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*
SYMBOL = SYSF       MODULES = 7, 19


 TXSLNK          2.3.1   78.244          01/00/00   00:02:18
LINK MAP

CONTROL FILE = DSC2:TEST1/LNK

LINKED OUTPUT FILE = DSC2:TEST1/SYS

LIST FILE = LP

OUTPUT FORMAT = ASCII

LIBRARIES

NO    ORGANIZATION    PATHNAME

1     SEQUENTIAL      :CBASIC/LIB
```

FIGURE E-4. LINK EDITOR LIST FILE OUTPUT (SHEET 2 OF 4)

PHASE 0, CBASIC ORIGIN = 0000 LENGTH = 1A50

MODULE	NO	ORIGIN	LENGTH	TYPE	DATE	TIME	CREATOR
STARTC	1	0000	0804	INCLUDE	01/02/79	14:08:03	SDSLNK
BAUDC	2	0804	0138	INCLUDE,1	01/03/79	09:51:17	SDSLNK
CVBDC	3	0C3C	034E	INCLUDE,1	01/03/79	09:52:32	SDSLNK
CVDB	4	0F8A	0182	INCLUDE,1	01/03/79	09:52:51	SDSLNK
CVGC	5	113C	005E	INCLUDE,1	01/03/79	09:53:06	SDSLNK
DIM	6	119A	00B8	INCLUDE,1	01/03/79	09:53:32	SDSLNK
FUNCC	7	1252	0056	INCLUDE,1	01/03/79	09:54:31	SDSLNK
GOSUB	8	12A8	00A0	INCLUDE,1	01/03/79	09:55:08	SDSLNK
IF	9	1348	0080	INCLUDE,1	01/03/79	09:55:24	SDSLNK
INPUT	10	13C8	0104	INCLUDE,1	01/03/79	09:55:39	SDSLNK
IPCOM	11	14CC	003A	INCLUDE,1	01/03/79	09:56:08	SDSLNK
JMP	12	1506	001C	INCLUDE,1	01/03/79	09:56:21	SDSLNK
LET	13	1522	0142	INCLUDE,1	01/03/79	09:56:32	SDSLNK
MOVE	14	1664	0060	INCLUDE,1	01/03/79	09:57:27	SDSLNK
POLY	15	16C4	0068	INCLUDE,1	01/03/79	09:57:51	SDSLNK
PRINTC	16	172C	0134	INCLUDE,1	01/03/79	09:58:21	SDSLNK
PUTB	17	1860	004E	INCLUDE,1	01/03/79	09:58:42	SDSLNK
SQRF	18	18AE	006C	INCLUDE,1	01/03/79	10:00:06	SDSLNK
ZZENDC	19	191A	000E	INCLUDE,1	01/03/79	10:01:02	SDSLNK
ROOT	20	1928	0128	INCLUDE			CONFIG

D E F I N I T I O N S

NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO
ABSF	128C	7	*ADDF	0984	1	ANDF	191A	19	ASCF	191A	19
ASR0	003C*	1	ASR1	003E*	1	ATNF	191A	19	B01	0961	1
B05	0965	1	B20	01FE	1	B3F	01FF	1	BASY	191A	19
BAUD	0B08	2	*BCRU	003A*	1	BITF	191A	19	BITY	191A	19
BMVE	1664	14	BUS	000C*	1	*C1	0960	1	C4	0962	1
C4A00	01FC	1	C6	096A	1	CCNT	0028*	1	CKEX	0586	1
*CLKADR	FE6C*	1	CLKI	191E	19	*CLKT01	FE72*	1	*CLKT02	FE74*	1
*CLKWS	FE6A*	1	CLLY	191A	19	COSF	191A	19	CRBF	191A	19
CRBY	191A	19	CRFF	191A	19	CRFY	191A	19	CVBD	0C48	3
*CVBE	0E4E	3	CVBI	0C40	3	CVC10	1164	5	CVCH	FEEC*	1
*CVDB20	10C4	4	CVDIFZ	0F8E	4	*CVDIZ	0F8A	4	CVGCN	113C	5
CVGCN1	1144	5	CVHD	FEF2*	1	CVHD01	FEF3*	1	CVHD15	FF01*	1
*DATX8	0201	1	*DDM	0022*	1	DEFY	191A	19	DIMY	119A	6
*DLC	0020*	1	DLIM	001E*	1	DS	FE76*	1	*DS1	FE7C*	1
*DS2	FF8C*	1	*DS3	FFC2*	1	*ERP	FEF2*	1	*EFLG	002A*	1
ELNM	002E*	1	ELSF	0030*	1	*ELSY	191A	19	ENUM	002C*	1
ERRY	191A	19	*EUS	0002*	1	EVAL	073C	1	EVALS2	0734	1
EVARZ	0582	1	EVERZ	05CC	1	*EVFX	05E4	1	*EVOP3A	087E	1
EVSDZ	0574	1	EVSFR	0714	1	*EVSKB	FEF2*	1	*EVSKE	FF94*	1
EXPF	191A	19	*FAD	0246	1	FADDI	051E	1	*FCLR	044E	1
*FDD	0482	1	FFLG	0032*	1	FIX	0A82	1	*FLDD	0234	1
*FLOAT	0300	1	*FMD	0362	1	*FNEG	02F6	1	*FNRM	0328	1
FNS	0004*	1	*FNSZ	000A*	1	FORY	191A	19	FPAC	FF9C*	1
FPAC2	FF9E*	1	FPAC4	FFA0*	1	*FPWP	FF9C*	1	*FSCL	0546	1
*FSD	023C	1	*FSRD	022C	1	FSUBI	0532	1	*FUNBK	16C4	15

FIGURE E-4. LINK EDITOR LIST FILE OUTPUT (SHEET 3 OF 4)

```
TXSLNK          2.3.1  78.244          01/00/00  00:02:18                    PAGE
NAME     VALUE NO   NAME     VALUE NO   NAME     VALUE NO   NAME     VALI.

FUNFX    16EE  15   *FUZZ    3B00* 1    *GETC    0BF6  2    GETCR    0BEA
GOS1     12F2  8    *GOS2    1306  8    GOS81    12A6  8    *GOSUN   12E6
GOSY     12B4  8    GOTY     12E6  8    GSC      0016* 1    GSS      0006
*GSSZ    0014* 1    HFLG     0034* 1    *HOUT    1822  16   IFLG     0036
IFY      1346  9    IMKY     1918  19   INPEND   14E0  11   INPF     126E
INPY     13C6  10   *INTFLG   FE56* 1    INTIN    191C  19   *INTSB   FE36
*INTSP   FE58* 1    *INTWP2  FE48* 1    IOB      0000* 1    IRTY     1916
*ISTICK  FE36* 1    *ITAB    FE16* 1    *ITAB2   FE18* 1    *JMPRO   150C
JPROA    1504  12   LANDF    1918  19   LEC      0024* 1    LENF     1918
LETY     1520  13   LINE     0160  1    LINE0    016E  1    LINE2    0178
LINE5    0160  1    LNOTF    1918  19   LNSZ     0084* 1    LOGF     1918
LORF     1918  19   LXORF    1918  19   MCHF     1918  19   MEMF     1918
MEMY     1918  19   MODE     0040* 1    *NIC     0064* 1    NKYF     0C08
NLIN     015A  1    NLIN0    0156  1    NOTF     1918  19   *NRV     0004
NVID     0012* 1    NVS      0014* 1    *NXTXB   0200  1    NXTY     1918
ONY      1918  19   ORF      1918  19   OUTL1    14CA  11   *PFIX    0ADC
PLC      001C* 1    PLF      0038* 1    *PLYX    1714  15   *PLYXX   1708
POPY     1334  8    POWF     1918  19   PRTEND   14D8  11   PRTY     172F
PUTB     185E  17   *RANDS   FE68* 1    RANDZ    1922  19   RANY     1918
RDDY     1918  19   *REST    0134  1    RNWY     1918  19   RODT     1924
RTNY     1312  8    SFSN     0AEE  1    SINF     1918  19   *SLN     0026
SLT      000E* 1    SQRF     18AC  18   SQRI     0004* 1    SRHF     1918
*SSP     FF76* 1    *SUBF    09DA  1    SYSF     1250  7    TEMP     FF94
*TEMP2   FF96* 1    *TEMP4   FF98* 1    *TEMP6   FF9A* 1    TICF     1918
TIMY     1918  19   TRPY     1918  19   TYO      0018* 1    TYPO     0B6C
*TYP11   0B6A  2    TYPB     0B7E  2    *TYPBE   0B7C  2    TYPC     0B74
*UCCNT   14F0  11   UFT      0008* 1    *UMSK    0003* 1    UNIT     001A
UNTY     1918  19   VDT      000A* 1    *VNT     0010* 1    WPR1     FE3A
*WPR103  FE90* 1    *WPR104  FE92* 1    *WPR108  FE9A* 1    WPR2     FFDC

19 WARNINGS

****  LINKING COMPLETED
```

FIGURE E-4. LINK EDITOR LIST FILE OUTPUT (SHEET 4 OF 4)

SOFTWARE TROUBLE REPORT

STR NUMBER _____
(for TI use) _____

SUBMITTED BY _____ DATE _____
COMPANY _____ TEL. NO. _____
MAILING ADDRESS _____
REPORTED TO _____ TEL. NO. _____

CUSTOMER INFORMATION:
SOFTWARE/DOCUMENT NAME _____
TEXAS INSTRUMENTS PART NO. _____ REVISION _____ VERSION _____
PROBLEM TYPE: (CIRCLE ONE)
SOFTWARE DOCUMENTATION OTHER (explain) _____
PROBLEM SUMMARY: _____

EQUIPEMNT DESCRIPTION:
MEMORY SIZE:
RAM _____
EPROM _____

	MANUFACTURER	MODEL
MICROCOMPUTER BOARD		
EXPANSION MEMORY BOARD(S)		
ADDITIONAL BOARDS IN SYSTEM		

PLEASE ATTACH ALL NECESSARY INFORMATION TO DUPLICATE THE ABOVE REPORTED DISCREPANCE. USE ADDITIONAL PAGES IF NECESSARY.

MAIL TO:

TEXAS INSTRUMENTS INCORPORATED
P.O. BOX 1443, M/S 6407
HOUSTON, TEXAS 77001

TEXAS INSTRUMENTS
INCORPORATED
Semiconductor Group
Post Office Box 1443 Houston, Texas 77001

MP318

Printed in U.S.A.