TEXAS INSTRUMENTS

9900

Family Software

TM 990 POWER BASIC REFERENCE MANUAL

MICROPROCESSOR SERIES

Copyright © 1978, Texas Instruments Incorporated

MP308 Rev. *A

TM 990 POWER BASIC REFERENCE MANUAL

TEXAS INSTRUMENTS
INCORPORATED

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time in order to improve design and to supply the best product possible.

TM990 POWER BASIC (Part Numbers: TM990/450, Evaluation POWER BASIC EPROMS and TM990/101M-10, Evaluation POWER BASIC module) software products are copyrighted by Texas Instruments Incorporated. All rights reserved. Property of Texas Instruments Incorporated.

In addition, TM990 POWER BASIC (Part Number: TM990/451, Development POWER BASIC and TM 990/452, Development Support option) software products are copyrighted by Texas Instruments Incorporated. This software may not be reproduced in any form for resale that is used in a software development system context without prior written permission from Texas Instruments Incorporated. However, source or object programs may be copied for any end-use application other than for a software development system.

"TM990 POWER BASIC Reference Manual" and "TM990 POWER BASIC Elementary Tutorial Manual" are copyrighted by Texas Instruments Incorporated. All Rights Reserved. Printed in the United States of America. 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.

Copyright, Texas Instruments Incorporated, 1978

PREFACE

This document describes Texas Instruments' POWER BASIC which is an interactive programming language used with the TM 990/101M or the TM 990/100M microcomputers.

The manual is organized into three major segments: Overview, Functional Description, and POWER BASIC Reference Guide. The first segment contains the introduction and a full discussion of the installation procedures. The second segment contains a functional description of POWER BASIC as well as general programming information that describes conventions of the language. The third segment is a more specific reference section which covers POWER BASIC in greater detail. It contains statement syntax and specific examples.

The following manuals present additional information relative to the use of POWER BASIC.

Hardware:

MP321 "TM 990/100M Microcomputer User's Guide"
MP337 "TM 990/101M Microcomputer User's Guide"
MP334 "TM 990/201 and TM 990/206 Expansion Memory Boards"
MP336 "TM 990/203 Expansion Memory Board"
MP344 "TM 990/302 Hardware User's Guide"

Software:

MP311 "TM 990 POWER BASIC Elementary Tutorial Manual"

TABLE OF CONTENTS

OVERVIEW

SECTION I. INTRODUCTION

1.1 General....1-1
1.1.1 Evaluation BASIC....1-1
1.1.2 Development BASIC....1-1
1.1.3 Development BASIC Enhancement Software Package....1-2
1.2 Interaction with POWER BASIC....1-2
1.3 Conventions Used in This Manual....1-2

SECTION II. INSTALLATION

2.1 POWER BASIC....2-1
2.2 Equipment Requirements....2-1
2.2.1 Microcomputer Board....2-2
2.2.2 Accessory Boards....2-2
2.2.3 Power Supply....2-2
2.2.4 Chassis....2-3
2.2.5 Terminal and Cables....2-3
2.2.6 Audio Cassettes and EPROM Programming...2-4
2.3 System Setup....2-4
2.3.1 Power Supply Connections....2-4
2.3.2 EPROM Insertion....2-7
2.3.3 CPU Jumper Settings....2-8
2.3.4 Five Switch Dip on TM 990/101M Board....2-13
2.3.5 Accessory Boards Jumper Settings....2-13
2.3.5.1 TM 990/302 Jumper and Switch Settings....2-13
2.3.5.2 TM 990/201 Jumper and Switch Settings....2-14
2.3.6 Board Insertion and Terminal Hookup....2-14
2.3.6.1 Board Insertion....2-15
2.3.6.2 Terminal Hookup....2-16
2.4 Operation....2-18
2.4.1 Verification....2-18
2.4.2 Power-up/Reset....2-18
2.5 Sample Programs....2-19
2.6 Debug Checklist....2-21
2.7 RAM Expansion....2-23
2.8 Cassette Transportation Operation....2-23
2.8.1 Procedures to Record on Digital Cassette From Development BASIC (SAVE)....2-23

2.8.2 Procedures to Record on Audio Cassette From Development BASIC (SAVE).....2-27
2.8.3 Procedures to Playback Tape From Digital Cassettes to POWER BASIC (LOAD).....2-28
2.8.4 Procedures to Playback Audio Cassette to Development BASIC (LOAD Unit #)....2-30
2.9 Two User or Single User/Two Partition Operation.....2-30
2.9.1 Single User/Two Partition POWER BASIC Configuration.....2-31
2.9.2 Two/User/Two Partition POWER BASIC Configuration.....2-33
2.9.3 Communications Between Multipartitions..2-34
2.9.4 Example.....2-34
2.10 EPROM Programming.....2-34
2.11 LOAD Vector.....2-35
2.12 TM 990/101M Second EIA Port.....2-35

FUNCTIONAL DESCRIPTION

SECTION III. GENERAL PROGRAMMING INFORMATION

3.1 General.....3-1
3.2 BASIC Language.....3-1
3.3 POWER BASIC Program.....3-1
3.4 Source Statement Format.....3-2
3.4.1 Character Set.....3-2
3.4.2 Line Number Field.....3-2
3.4.3 Statement Field.....3-3
3.4.4 Tail Remark.....3-3
3.5 Edit Mode Commands.....3-3
3.6 Constants.....3-5
3.6.1 Hexadecimal Integer Constants.....3-5
3.6.2 Decimal Integer Constants.....3-5
3.6.3 Decimal Real Constants.....3-5
3.6.4 String Constants.....3-6
3.7 Variables.....3-6
3.7.1 Simple Variables.....3-7
3.7.2 Numeric Array Variables.....3-7
3.7.3 Simple String Variables.....3-7
3.7.4 String Array.....3-8
3.7.5 Reserved Variables.....3-9
3.7.6 Variable Storage.....3-9
3.7.6.1 Number Array Storage.....3-9
3.7.6.2 Strings and String Array Storage.....3-11
3.7.7 Variable Format and Accuracy.....3-12
3.8 Operators and Expressions.....3-15
3.8.1 Arithemetic Operators.....3-15

3.8.2 Arithmetic Expressions.....3-15
3.8.3 Logical Operators.....3-16
3.8.4 Logical Expressions.....3-16
3.8.5 Relational Operators.....3-16
3.8.6 Boolean Operators.....3-17
3.8.7 Boolean and Relational Expressions.....3-17
3.8.8 Expression Evaluation.....3-18
3.9 Multiple Statements "::".....3-18
3.10 Keyboard Mode.....3-19
3.11 Errors and Error Listings.....3-20
3.12 Reset and Load Function Operation.....3-23

POWER BASIC REFERENCE GUIDE

SECTION IV. BASIC COMMANDS

4.1 General.....4-1
4.2 CONTINUE Command.....4-1
4.3 LIST Command.....4-2
4.4 LOAD Command.....4-2
4.5 NEW Command.....4-5
4.6 PROGRAM Command.....4-6
4.7 RUN Command.....4-11
4.8 SAVE Command.....4-12
4.9 SIZE Command.....4-13

SECTION V. BASIC STATEMENTS

5.1 General.....5-1
5.2 COMMENT or Remark (REM) Statement.....5-3
5.3 Dimension Statement.....5-4
5.4 Function Definition.....5-5
5.5 Variable Assignment.....5-6
5.5.1 LET Statement.....5-6
5.6 Control and Computed Transfer Statements....5-7
5.6.1 Unconditional GOTO Statement.....5-7
5.6.2 Conditional IF-THEN-ELSE Statement.....5-8
5.6.2.1 IF-THEN Statement.....5-8
5.6.2.2 ELSE Statement.....5-9
5.6.3 Subroutine (GOSUB, POP, and RETURN) Statements.....5-10
5.6.4 ON Statement.....5-15
5.6.5 FOR/NEXT Loops.....5-15
5.6.6 ERROR Statement.....5-20
5.6.7 STOP Statement.....5-21
5.6.8 END Statement.....5-22
5.7 Internal Input Statements.....5-22
5.7.1 DATA Statement.....5-23

5.7.2 READ Statement.....5-23
5.7.3 RESTOR Statement.....5-24
5.8 Terminal I/O Statements.....5-26
5.8.1 INPUT Statement.....5-31
5.8.2 PRINT Statement.....5-31
5.8.2.1 Print Formatting.....5-36
5.8.2.2 TAB.....5-42
5.8.2.3 Summary - Print Statement Rules.....5-43
5.8.3 UNIT Statement.....5-44
5.8.4 BAUD Statement.....5-45
5.9 Interrupt Processing.....5-46
5.9.1 IMASK Statement.....5-47
5.9.2 TRAP Statement.....5-47
5.9.3 IRTN Statement.....5-48
5.9.4 Assembly Language Processors.....5-48
5.10 BASE Statement.....5-50
5.11 TIME Statement.....5-51
5.12 RANDOM Statement.....5-53
5.13 ESCAPE and NOESCAPE Statements.....5-54
5.14 CALL Statement.....5-55

SECTION VI. CHARACTER STRINGS

6.1 General.....6-1
6.2 Character Assignment.....6-1
6.3 Character Concatenation.....6-3
6.4 Character Pick.....6-4
6.5 Character Replacement.....6-4
6.6 Character Insertion.....6-5
6.7 Character Deletion.....6-5
6.8 Byte Replacement.....6-6
6.9 Convert ASCII Character to Number.....6-6
6.10 Convert Number to ASCII Character.....6-7
6.11 String Length Function.....6-8
6.12 Character Search Function.....6-8
6.13 Character Match Function.....6-9
6.14 ASCII Character Conversion Function.....6-10

SECTION VII. POWER BASIC FUNCTIONS

7.1 General.....7-1
7.2 Mathematical Functions.....7-1
7.2.1 Absolute Value Function (ABS).....7-1
7.2.2 Arctangent Function (ATN).....7-2
7.2.3 Sine and Cosine Functions (SIN)(COS)...7-2
7.2.4 Exponential Function (EXP).....7-2
7.2.5 Integer Part Function (INP).....7-3
7.2.6 Logarithm Function (LOG).....7-3
7.2.7 Square Root Function (SQR).....7-4

7.3 String Functions.....7-4
7.3.1 ASCII Character Conversion Function....7-4
7.3.2 Length Function (LEN).....7-5
7.3.3 Character Match Function (MCH).....7-5
7.3.4 Character Search Function (SCH).....7-6
7.4 Miscellaneous Functions.....7-6
7.4.1 CRU Single Bit Function (CRB).....7-6
7.4.2 CRU Field Function (CRF).....7-7
7.4.3 Key Function (NKY).....7-7
7.4.4 System Interrogation (SYS) Function....7-7
7.4.5 Delta Time (TIC) Function.....7-8
7.4.6 Memory Modification (MEM) Function.....7-8
7.4.7 Bit Modification (BIT) Function.....7-9
7.4.8 Random Number (RND) Function.....7-10

APPENDICES

Appendix A Error Codes..... A-1
Appendix B Statement and Command Summary.....B-1
Appendix C Sample Programs.....C-1
Appendix D Xops.....D-1

LIST OF ILLUSTRATIONS

Figure 2-1 TM 990/101M Memory Placement.....2-5
Figure 2-2 TM 990/100M Memory Placement.....2-6
Figure 2-3 TM 990/101M Jumper Locations.....2-10
Figure 2-4 TM 990/100M Jumper Locations.....2-11
Figure 2-5 743 KSR Terminal Hookup.....2-13
Figure 2-6 Connector P2 Connected to RS-323-C Device (Model 733 ASAR).....2-13
Figure 2-7 Connector P2 Connected to TTY Device....2-14
Figure 2-8 Power Supply Hookup.....2-15
Figure 2-9 TM990/101M Board in TM 990/510 Chassis..2-16
Figure 2-10 Evaluation BASIC Memory Maps.....2-27
Figure 2-11 ASR Module Assembly Switch Panel.....2-28

Figure 5-1 GOSUB Example.....5-13

LIST OF TABLES

Table 2-1 TM 990/101M Jumper Settings.....2-6
Table 2-2 TM 990/100M Jumper Settings.....2-7
Table 2-3 TM 990/101M Board Jumper Positions.....2-8
Table 2-4 TM 990/100M Board Jumper Positions.....2-9
Table 2-5 5-Bit Dip Switch Option.....2-21
Table 2-6 Recommended RAM Expansion Configurations...2-26

Table 5-1 POWER BASIC Statements.....5-2
Table 5-2 Formatting String Characters.....5-41
Table 5-3 Interrupt Level Data.....5-50

OVERVIEW

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 EPROMs, TM 990 boards, and floppy diskettes. The family is primarily designed to support the industrial user, providing not only classical features often found in BASIC, but also features specifically designed to support real time industrial control applications. Some family members provide capability for supporting either multiusers or concurrent tasks written in POWER BASIC; other members support single user environments. Typical applications for the language include data acquisition and control, data communications, direct digital control, data reduction and analysis, etc. The two family members documented in this manual are Evaluation and Development Basic. See Appendix B for an enumeration of the language features supported by each of the products.

1.1.1 EVALUATION BASIC

Evaluation BASIC requires the TM 990/100M or the TM 990/101M CPU board and optionally a TM 990/201, 206, or 203 memory expansion board. Evaluation BASIC supports one or two users on RS-232-C compatible terminals including the TI 733. If desired, Evaluation BASIC has the option of two partitions for the single user. Section II discusses this feature and provides a full discussion of the installation procedures for Evaluation BASIC.

1.1.2 DEVELOPMENT BASIC

Development BASIC requires the TM 990/100M or the TM 990/101M CPU board; also required is the TM 990/302 software development board or the TM 990/201 memory expansion board. Optionally, additional TM 990/201, 206, or 203 memory expansion boards may be used. Development BASIC will support a single user on an RS-232-C compatible terminal, including the TI 733. Development BASIC supports a superset of the Evaluation BASIC statements and commands, as well as the capability to link to the Development BASIC Enhancement Software Package as explained in paragraph 1.1.3. Section 2 provides a full discussion of the installation procedures for Development BASIC.

*Trademark of Texas Instruments

1.1.3 DEVELOPMENT BASIC ENHANCEMENT SOFTWARE PACKAGE

The Development BASIC Enhancement Software Package is used in conjunction with the Development BASIC Package. The software resides on either the TM 990/302 software development board or the TM 990/201 memory expansion board. The Development BASIC Enhancement Software Package provides additional capabilities that allow the user to save application programs on audio cassette, program these applications into TMS2716 EPROM's, provide decimal print formatting using a string image, and provide complete error message reporting.

These additional features are described in detail in the appropriate sections of this manual. Section 2 provides a full discussion of the installation procedures for the Enhancement Software Package.

1.2 INTERACTION WITH POWER BASIC

Interaction with BASIC involves user input of a series of program statements and commands and user response to program-generated requests for input. The user may enter program statements or invoke commands required to examine, debug, or run the program. The user completes each statement or command by entering a carriage return.

The carriage return terminates and enters the line, advances one line, and waits for further keyboard input. Each program statement is stored as it is entered, and the program may be listed at any time during its generation or at its completion by using the LIST command. Commands are not stored but are executed when they are entered. At any time, the user may halt execution or terminate a statement or command by striking the escape (or BREAK) key.

1.3 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>
```

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

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

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

INSTALLATION

2.1 POWER BASIC

This section describes the setting-up and initial operation of both the Evaluation and the Development BASIC software systems. The installation and operation steps described are common to both software packages except as noted.

Evaluation BASIC is shipped in one of two forms:

- A TM 990/101M-10 microcomputer board with Evaluation Basic resident in EPROM's and inserted in the board as initially shipped from the factory

- A TM 990/450 software package consisting of a set of four EPROM's containing Evaluation BASIC, which are to be inserted into the user supplied TM 990/101M or TM 990/100M microcomputer board.

Development BASIC is shipped as:

- A TM 990/451 software package consisting of six EPROM's containing Development BASIC, which are to be inserted into the user supplied TM 990/101M or TM 990/100M microcomputer board, and TM 990/302 software development board or TM 990/201 EPROM/RAM expansion board.

- A TM 990/452 Development BASIC Enhancement Software Package consisting of a pair of EPROMs which are to be inserted into the TM 990/302 or the TM 990/201 board.

It is recommended that the microcomputer board have a fully populated RAM area (4K bytes on TM 990/101M or 1K bytes on TM 990/100M) to provide sufficient user storage for significant POWER BASIC program development.

2.2 EQUIPMENT REQUIREMENTS

Both the equipment required and appropriate options are described in the following paragraphs. The different equipment requirements for Evaluation BASIC and Development BASIC are also detailed.

2.2.1 MICROCOMPUTER BOARD

One of the following microcomputer boards will be required:

- TM 990/101M (-1, -2 or, -3) microcomputer board
- TM 990/100M (-1, -2, or -3) microcomputer board

Or for Evaluation BASIC only:

- TM 990/101M-10 microcomputer board shipped from the factory containing the Evaluation BASIC EPROM set

2.2.2 ACCESSORY BOARDS

To make use of all the features of the Development BASIC Enhancement Software Package, it is necessary to include the TM 990/302 Software Development board in the system configuration. If the EPROM programming and audio cassette capabilities of the Enhancement Software Package are not required, or if the Enhancement Software is not being used, then the system may be configured with either a TM 990/302 Software Development board or a TM 990/201 (-41, -42, or -43) Memory Board.

2.2.3 POWER SUPPLY

The power requirements of the boards that may be used for system configuration (see paragraphs 2.2.1 and 2.2.2) are listed below in Table 2-1. The power supply used must be capable of supplying the total required current of the selected system configuration as a minimum. It is recommended that current ratings of the power supply be increased above the minimum to allow for the addition of other boards to the system (memory, I/O, etc.). Regulation must be ±3% on all supplies except the +48V*.

TABLE 2-1.

TM 990 CIRCUIT BOARD CURRENT REQUIREMENTS

TM 990 Circuit board	Current Requirements			
	+5V	+12V	-12V	+48V*
/100M CPU Board	1.4	0.2	0.1	
/101M CPU Board	2.6	0.5	0.4	
/302 Software Devel. Board	0.9	0.2	0.1	0.1
/201-41	2.5	0.2	0.5	
/201-42	3.0	0.4	0.6	
/201-43	5.5	0.8	0.7	
*35-55V unregulated for EPROM Programming All currents are MAX				

2.2.4 CHASSIS

The use of a TM 990/510 4 slot chassis, TM 990/520 8 slot chassis, or an equivelent is necessary for the set-up of Development BASIC since more than one board is required in the system, configuration. Evaluation BASIC can be executed on a single board system, so a chassis is not a requirement but does facilitate set-up and use of the system.

Alternately, one of 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 225-804-50
- VIKIN 3VH50/9CND5
- ELCO 00-6064-100-061-001

2.2.5 TERMINAL AND CABLES

POWER BASIC supports the following terminal devices:

- RS-232-C compatible terminal (using a TM 990/502 cable), or the TI ASR 733 (using a TM 990/505 cable): see Appendix B of the "TM 990/100M" or "TM 990/101M 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 "TM 990/100M or "TM 990/101M Microcomputer User's Guide" for special cabling requirements. The TM 990/503 cable may be used to interface a 743/745 terminal.

- Teletype Model 3320/5JE (for TM 990/100M-1, TM 990/101M-1, and -3 microcomputer boards only): see Appendix A of the "TM 990/100M" of "TM 990/101M Microcomputer User's Guide" for required modifications for 20 mA neutral current loop operation and proper cable connections to the TM 990/504 cable.

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 "TM 990/100M" or "TM 990/101M Microcomputer User's Guide" for the signal configuration used by the main I/O port.

2.2.6 AUDIO CASSETTES AND EPROM PROGRAMMING

If the sytsem is configured with the TM 990/302 and the TM 990/452 Enhancement Software Package, the user has the option of operating with audio cassettes and/or EPROM programming. The following cassette players are recommended for correct operation:

- Realistic CTR-40
- Sharp RD 610
- Sears Model No. 799.21683700
- Bigston KD 130
- Panasonic RQ 413AS

Criterion for selection of audio cassette recorder:

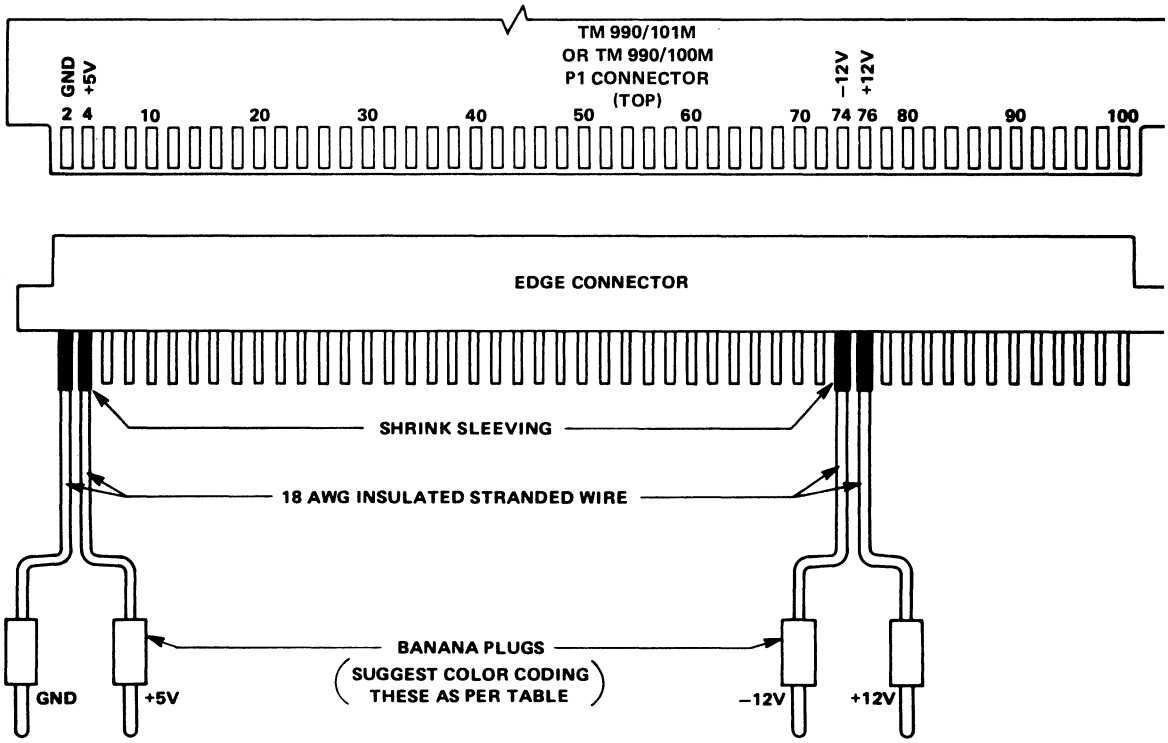
- UL approved
- Ear, aux, and remote inputs
- Volume control
- AC operation

2.3 SYSTEM SETUP

This section describes the steps required to set-up the system to begin operation.

2.3.1 POWER SUPPLY CONNECTIONS

Figures 2-1 and 2-2 illustrate power supply hookup by connection to a lone 100 pin edge connector and by installation in a card chassis, respectively. Only the TM 990/101M microcomputer board is displayed in the figures. However, the figures are applicable to both the TM 990/100M and the TM 990/101M microcomputer boards.



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.

0001417

FIGURE 2-1. POWER SUPPLY HOOKUP

Figure 2-1 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 2-1 shows suggested color coding for the power supply plugs. To prevent incorrect connection, label the top side 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 2-2. The correct voltages should also be verified at the chassis or edge connector prior to insertion of a board.

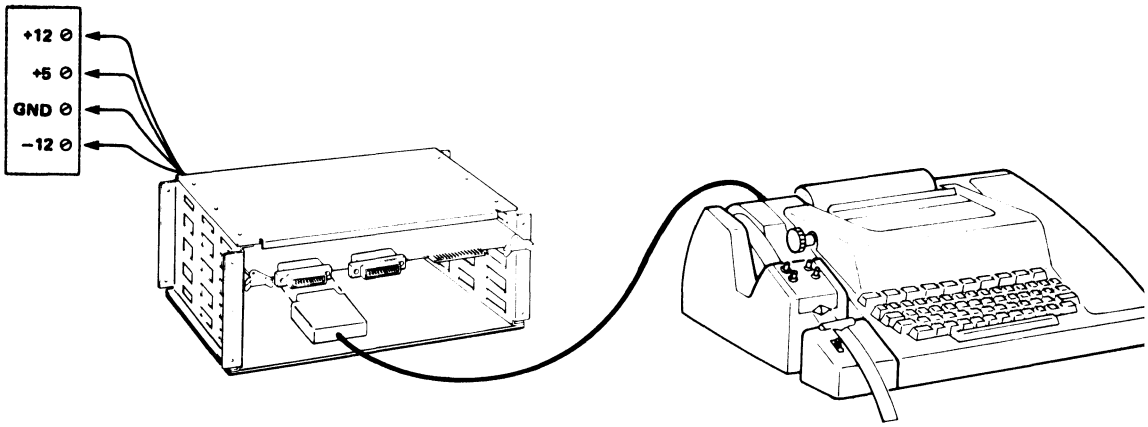


FIGURE 2-2. TM990/101M BOARD IN TM 990/510 CHASSIS

Power connection to the second board in a Development BASIC System is through the system backplane. When the system is configured with a TM 990/302 Software Development board and a TM 990/452 Enhance Software Package, the user may program TMS2716 EPROMs. This requires that the programming voltage (35-55V) be connected to pins 1 and 2 of TB1 on the TM 990/302. Pin 2 of TB1 is the positive. Care should be taken to guard against ground violations.

2.3.2 EPROM INSERTION

Skip this step and continue on to paragraph 2.3.3 if you have received the TM 990/101M-10 microcomputer board with the Evaluation BASIC EPROM's already installed.

Figures 2-1 and 2-2 show EPROM and RAM Memory locations on the TM 990/101M and TM 990/100M, respectively. The user should carefully remove all EPROM chips from the board (marked as U42, U43, U44, and U45) if present, and place them in conductive foam for safe keeping. Note that the POWER BASIC EPROM's in the plastic shipping case are appropriately marked as U42, U43, U44, and U45, and that each EPROM is marked with the part number (TM 990/450 or TM 990/451). Remove these EPROM's one at a time beginning with U42, and carefully insert them into the board sockets. Make sure they are placed in the correct sockets and that pin (1 next to white dot or EPROM) 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. Development BASIC consists of six TMS2716 EPROMs, all of which will be marked TM 990/451. Four of the six will also be marked U42 through U45, corresponding to the TM 990/100M or TM 990/101M EPROM sockets. The two other EPROM's marked as U15 and U17 need to be inserted in either the TM 990/302 or the TM 990/201 EPROM sockets. If the TM 990/302 is used, the EPROM (marked U15) should be inserted in the socket marked U15, and the EPROM (marked U17) should be inserted in the socket (marked U17). If the TM 990/201 is used, the EPROM (marked U15) should be inserted in the socket marked U64, and the EPROM (marked U17) is inserted in the socket marked U56.

The Development BASIC Enhancement Software Package (TM 990/452) consists of two TMS2716 EPROMs. Both EPROMs are marked with the part number and an additional marking (U14 or U16). The EPROM (marked U14) is inserted into the TM 990/302 socket marked U14 or the TM 990/201 socket marked U65, while the one (marked U16) is inserted into the TM 990/302 socket marked U16 or the TM 990/201 socket marked U57.

2.3.3 CPU JUMPER SETTINGS

The user should appropriately connect and verify that the jumper configurations on the TM 990/101M or TM 990/100M board are as described in Table 2-2 and Table 2-3 respectively.

Tables 2-4 and 2-5 provide detailed jumper placement information for use if the CPU board does not have the required jumper placement. Figure 2-3 and 2-4 show the board locations of these jumpers for reference.

TABLE 2-2

TM 990/101M 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 2-3

TM 990/100M 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

TABLE 2-4

TM 990/101M 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	E29-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-#52	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 TM 990/101M-1 and -3 only

** On TM 990/101M-2 only

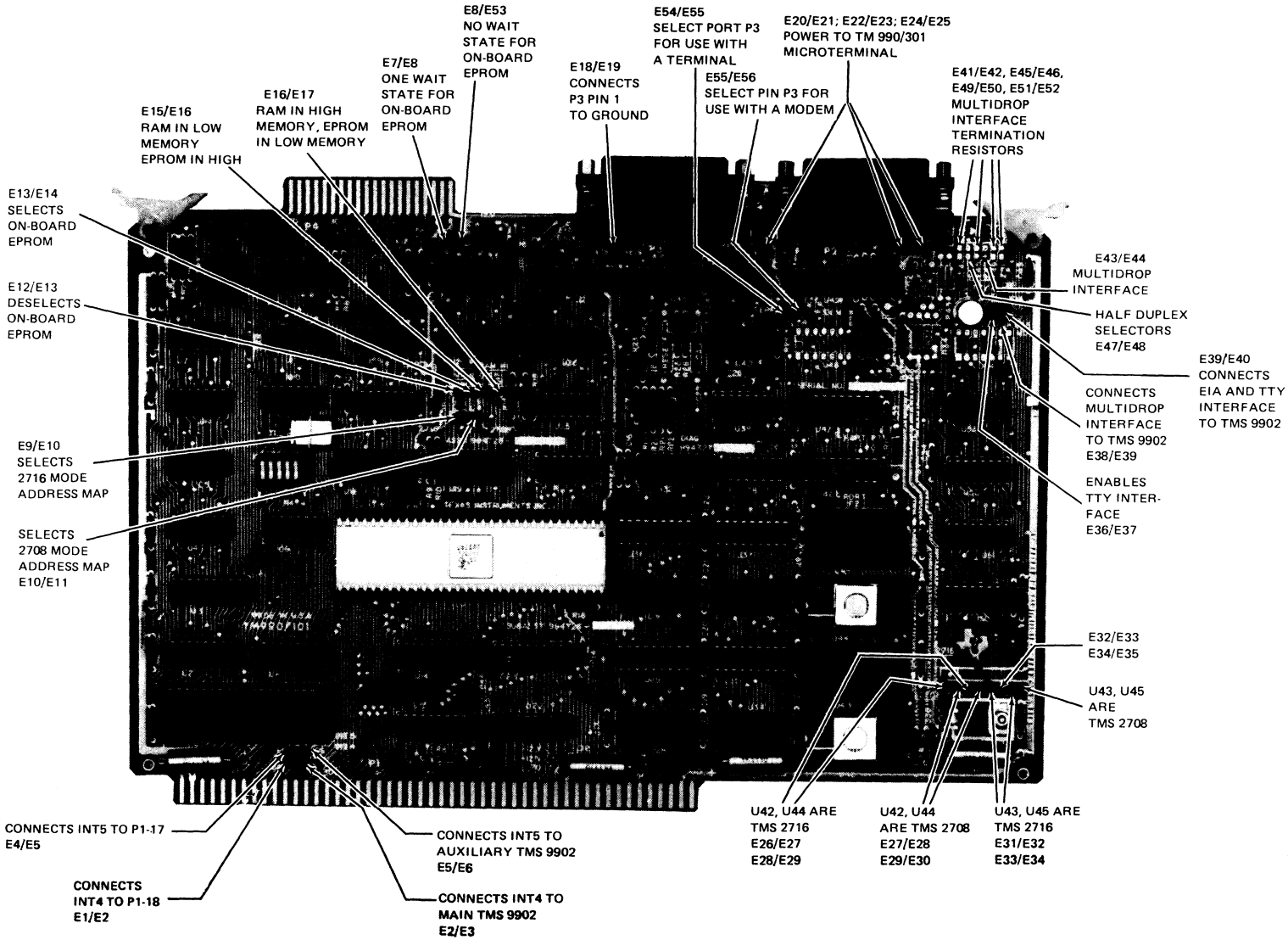
Note 1: These jumpers should be removed since the TM 990/301 microterminal is not used. (May be left installed for certain terminals, e.g., TI 743.)

TABLE 2-5

TM 990/100M 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 TM 990/301 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


E8/E53
NO WAIT
STATE FOR
ON-BOARD
EPROM
E54/E55
SELECT PORT P3
FOR USE WITH
A TERMINAL
E20/E21; E22/E23; E24/E25
POWER TO TM 990/301
MICROTERMINAL
E7/E8
ONE WAIT
STATE FOR
ON-BOARD
EPROM
E18/E19
CONNECTS
P3 PIN 1
TO GROUND
E55/E56
SELECT PIN P3 FOR
USE WITH A MODEM
E41/E42, E45/E46,
E49/E50, E51/E52
MULTIDROP
INTERFACE
TERMINATION
RESISTORS
E16/E17
RAM IN HIGH
MEMORY, EPROM
IN LOW MEMORY
E15/E16
RAM IN LOW
MEMORY
EPROM IN HIGH
E13/E14
SELECTS
ON-BOARD
EPROM
E43/E44
MULTIDROP
INTERFACE
E12/E13
DESELECTS
ON-BOARD
EPROM
HALF DUPLEX
SELECTORS
E47/E48
E39/E40
CONNECTS
EIA AND TTY
INTERFACE
TO TMS 9902
CONNECTS
MULTIDROP
INTERFACE
TO TMS 9902
E38/E39
E9/E10
SELECTS
2716 MODE
ADDRESS MAP
ENABLES
TTY INTER-
FACE
E36/E37
SELECTS
2708 MODE
ADDRESS MAP
E10/E11
E32/E33
E34/E35
U43, U45
ARE
TMS 2708
CONNECTS INT5 TO P1-17
E4/E5
CONNECTS
INT4 TO P1-18
E1/E2
CONNECTS INT5 TO
AUXILIARY TMS 9902
E5/E6
CONNECTS INT4 TO
MAIN TMS 9902
E2/E3
U42, U44 ARE
TMS 2716
E26/E27
E28/E29
U42, U44
ARE TMS 2708
E27/E28
E29/E30
U43, U45 ARE
TMS 2716
E31/E32
E33/E34

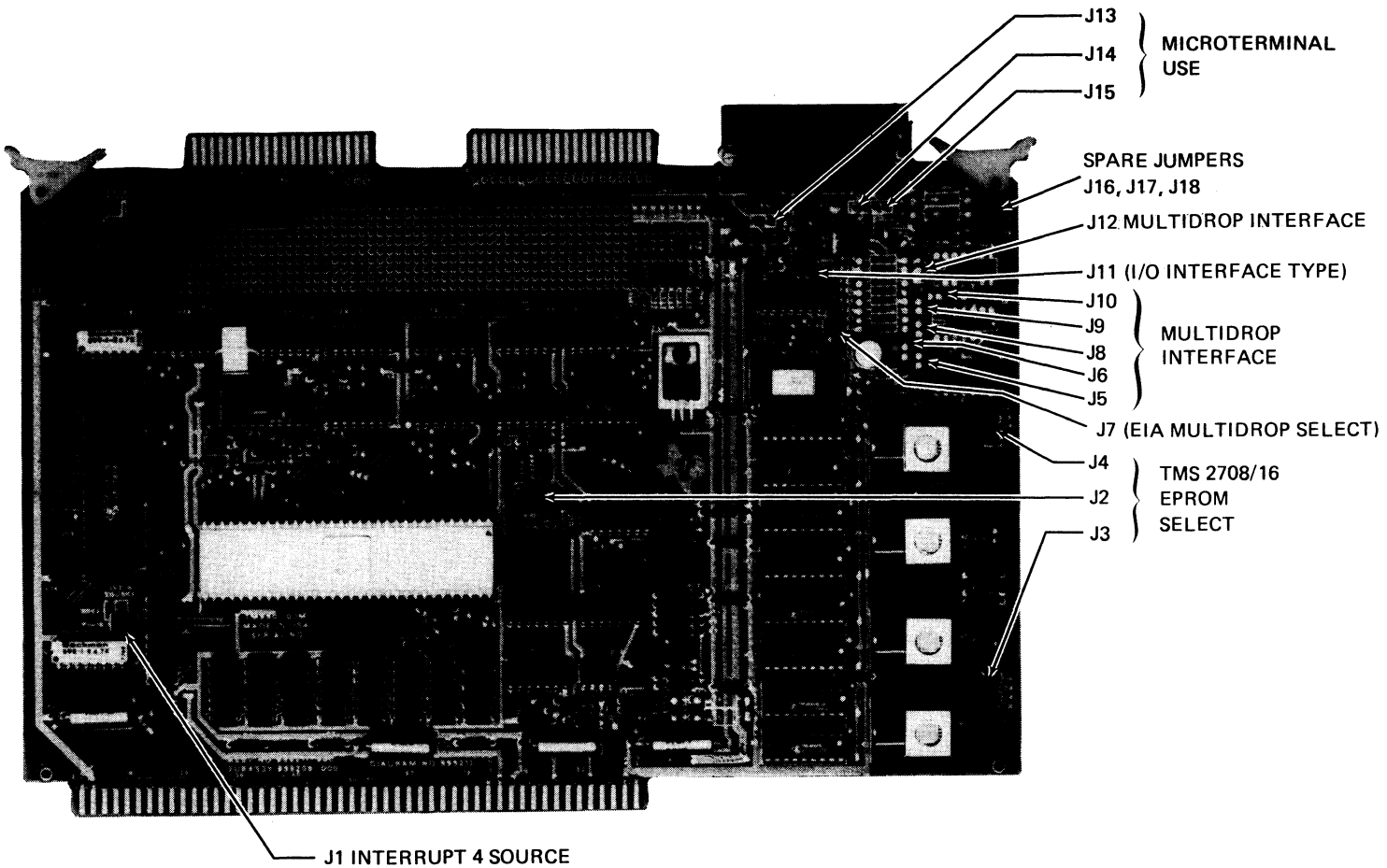


FIGURE 2-4. TM 990/100M JUMPER LOCATIONS

2.3.4 FIVE SWITCH DIP ON TM 990/101M BOARD

Evaluation BASIC on the TM 990/101M board is capable of supporting a two user Power BASIC System using ports P2 and P3, or a single user/ two-partition Power BASIC system using only port P2. For this partitioning of memory between users or between user partitions is required. It is performed by reading the positions of the five-switch dip on the TM 990/101M board. (Note that reading the CRU lines on the TM 990/100M which correspond to the CRU lines of the dip switches results in only single user operation.) For initial power-up, the user should have all swtiches of the five-switch dip in the OFF position. This configures Evaluation BASIC for single user, single partition operation. The CRU lines corresponding to the DIP switch are never addressed by Development BASIC, so the switch settings are immaterial.

2.3.5 ACCESSORY BOARDS JUMPER SETTINGS

This section describes the necessary switch and jumper settings for the second board in a Development BASIC system configuration. The second board can be either a TM 990/201 or a TM 990/302 board. Only the section giving details on the selected board need be read.

2.3.5.1 TM 990/302 JUMPER AND SWITCH SETTINGS

There are two stake pin jumpers on the TM 990/302; wait/no wait and load/no load. The TMS2716 EPROMs shipped have a 450 ns access time and require one wait state: the wait/no wait should be set for "wait" (E1 connected to E2). Since Development BASIC operation is initiated by the RESET stimulus, the TM 990/302 LOAD logic should be disabled by setting the load/no load jumper to "no load" (E5 connected to E6).

The DIP platform is used to configure the TM 990/302 for the type of EPROM being used. Development BASIC is shipped in TMS2716 EPROMs and requires wiring of the platform as shown in Figure 2-5.

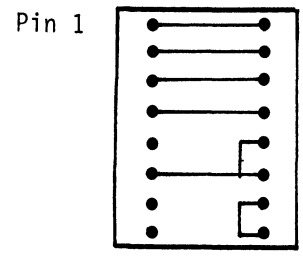


Figure 2-5. TM 990/302 XU25 PLATFORM WIRING

The DIP switch on the TM 990/302 board is used to select where the EPROM and RAM appear in the system memory map. Development BASIC requires the on board EPROM to be from 2000_{16} to $3FFF_{16}$. This mapping is achieved by setting SW1, SW2, and SW3 to OFF ON OFF respectively.

2.3.5.2 TM 990/201 JUMPER AND SWITCH SETTINGS

Development BASIC requires the EPROM to be addressed from 2000_{16} to $2FFF_{16}$ (2000_{16} to $3FFF_{16}$ with the Enhancement Software Package). The DIP switch settings to accomplish this are listed in Table 2-6; also shown are the switch settings to place the TM 990/201 RAM from $EFFF_{16}$ down.

2.3.6 BOARD INSERTION AND TERMINAL HOOK-UP

These procedures assume that the POWER BASIC EPROM's are resident in the required address space as described in paragraphs 2.3.2 and that a terminal and cable of the proper type to match the intended serial interface are also employed.

CAUTION

Be very careful to apply the correct voltage levels to the TM 990 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 with power applied. This is also true for front edge connections (cassette cables or EPROM personality cards).

Texas Instruments assumes no responsibility for damage caused by improper wiring or voltage application by the user.

TABLE 2-6 TM 990/201 SWITCH SETTINGS
EPROM - 2000_{16} to $3FFF_{16}$
RAM - $EFFF_{16}$ $DOWN_{16}$

Memory Board	SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8
TM 990/201-41	ON	ON	OFF	OFF	OFF	ON	OFF	OFF
TM 990/201-42	ON	ON	OFF	OFF	ON	ON	ON	OFF
TM 990/201-43	ON	ON	OFF	OFF	ON	ON	ON	ON

2.3.6.1 BOARD INSERTION

Figure 2-6 shows how to correctly place the microcomputer board in the TM 990/510 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 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 Development BASIC system should be inserted in the next slot down, although this is not critical.

Since the Evaluation BASIC can be used as a single board system, a lone edge connector can be used (as described in paragraph 2.3.1) in this configuration. The same caution should be used in voltage level checking each time the board is inserted.

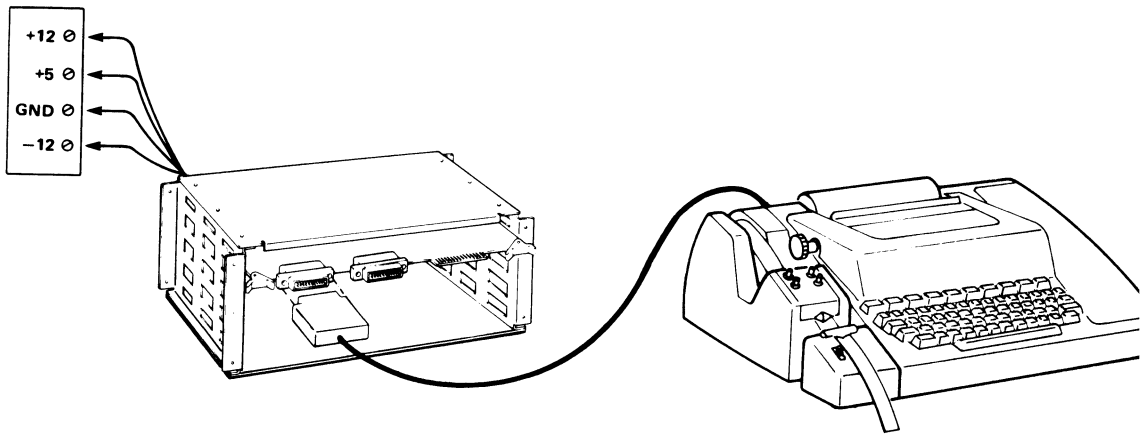


FIGURE 2-6. TM 990/101M BOARD IN TM 990/510 CHASSIS

2.3.6.2 TERMINAL HOOKUP

Figure 2-7 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 2-7. Figure 2-8 shows a RS-232 terminal (e.g., TI 733), and Figure 2-9 shows a TTY, connected to the TM 990/101M 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 TM 990/101M" or "TM 990/100M Microcomputer User's Guide". Evaluation BASIC configured as single user must have its terminal device connected to the main communications port (connector P2) at the corner of the board.

POWER BASIC operates the EIA/TTY ports at any of the following baud rates:

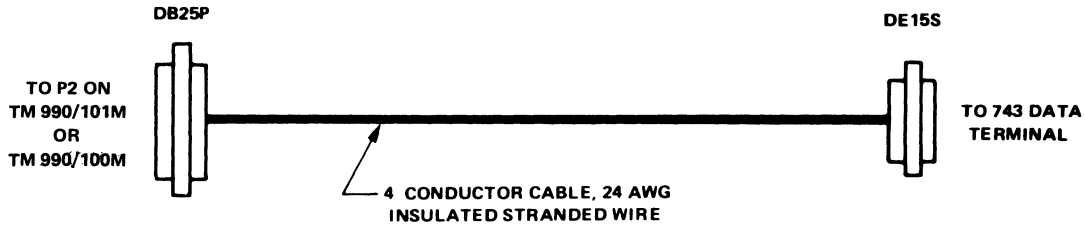
110, 300, 1200, 2400, 4800, 9600 or 19200 baud.

There is a 200 ms delay following a carriage return for all baud rates at or below 1200 baud. The delay allows for printhead travel.

The TMS9902 asynchronous communication controller is initialized by Evaluation and Development BASIC for a seven-bit ASCII character, even parity, and two stop bits (for compatibility with all terminals). At the terminal, set the baud rate of the terminal to one of the above speeds.

Power BASIC also uses conversational mode 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 the printer of the TI 733 terminal will operate with Evaluation BASIC only at or below 300 baud. Therefore, do NOT use the 733 terminal set to high speed or 1200 baud for communication with Evaluation BASIC. TI 733 ASR 1200 baud operation is allowed with Development BASIC, with some restrictions, see the note in paragraph 2.8.3



CONNECTIONS		
PIN ON DE15S	PIN ON DB25P	SIGNAL
13	2	XMIT
12	3	RECV
11	8	DCD
1	7	GND

A0001418

FIGURE 2-7. 743 KSR TERMINAL HOOKUP

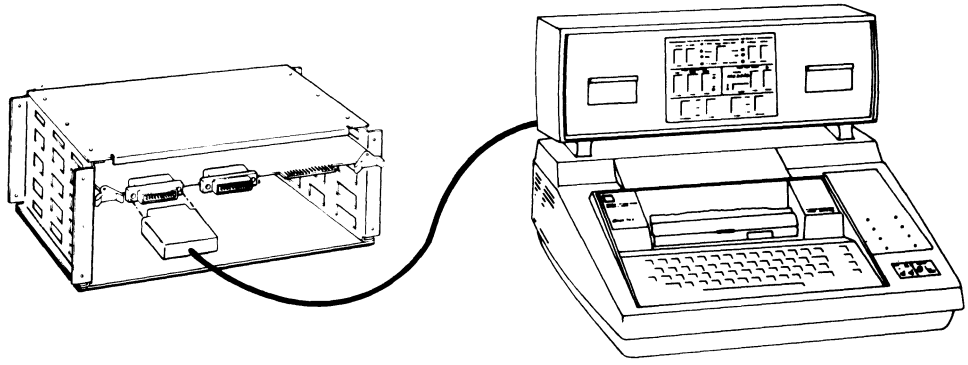


FIGURE 2-8. CONNECTOR P2 CONNECTED TO RE -232-C DEVICE (MODEL 722 ASR)

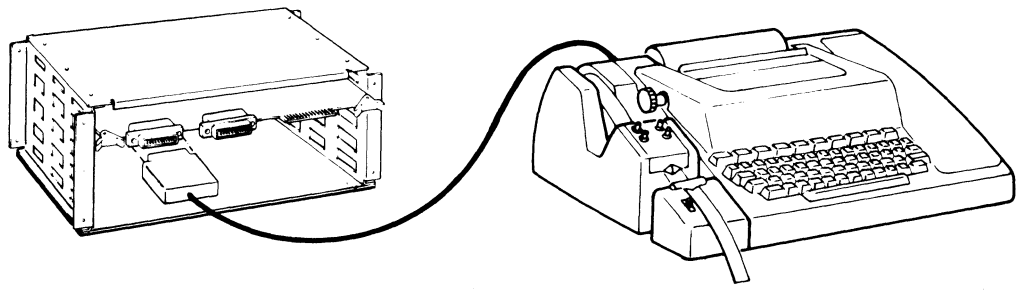


FIGURE 2-9. CONNECTOR P2 CONNECTED TO TTY DEVICE

2.4 OPERATION

2.4.1 VERIFICATION

Verify the following conditions before applying power:

- Power connected to correct pins on P1 connector
- Terminal cable between P2 connector (NOT P3 for single user) and terminal
- Jumpers in correct positions (see Paragraph 2.3.3)
- Five-switch dip all in OFF position
- Baud rate and communications mode are correctly set at terminal and terminal is ON LINE

2.4.2 POWER-UP/RESET

To power-up and initialize POWER BASIC, perform the following sequence:

a) Apply power to board and data terminal.
b) Activate the RESET switch near the corner of the microcomputer board. This causes BASIC to begin execution.
c) Press the "A" or carriage return key on the terminal device. BASIC measures the time of the start bit and determines the baud rate. A carriage return time delay of 200 ms will be provided for all baud rates at or slower than 1200 baud.

d) BASIC prints the banner message and, on a new line, the "*READY" message as shown below.

```
TM990 BASIC REV X.n.m
*READY
```

Where:

X=language level
n=the release number
m=the revision number

At this point, POWER BASIC is in keyboard mode awaiting keyboard input of POWER BASIC commands or statements.

2.5 SAMPLE PROGRAMS

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

When Evaluation BASIC powers up it displays the following banner message. The user may then enter 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).

```
TM990 BASIC REV. X.n.m
*READY
SIZE
PRGM: 18 BYTES
VARS: 0 BYTES
FREE: 3564 BYTES
      (SIZE numbers will depend on system configuration)
```

The user may then enter the following program:

```
 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 ROOT 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: 292 BYTES
VARS: 0   BYTES
FREE: 3272 BYTES
LIST
 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 ROOT 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 then 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 then enter the SIZE command to display the program size and the variable storage used by the program.

```
SIZE
PRGM: 244 BYTES
VARS:  28 BYTES
FREE: 3244 BYTES
```

All variable and program space may then be cleared as shown by the following sequence:

```
NEW
TM990 BASIC REV x.x.x
*READY
```

```
SIZE
PRGM:  18 BYTES
VARS:  0 BYTES
FREE:  3567 BYTES
```

2.6 DEBUG CHECKLIST

If the microcomputer does not respond correctly, turn the power OFF. Do not turn the power ON again until you are reasonably sure that 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 circuits:
  - Proper cable hookup to P2 connector and to terminal. Verify with data in Appendices A and B of the "TM 990/101M" or "TM 990/100M 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. (Recall that Evaluation BASIC will not communicate with the 733 ASR at 1200 baud.)

- Check jumper plug positions against Table 2-2 or 2-3.

- Be sure BASIC EPROM's are in place correctly.

- Be sure all switches of five position dip switch are in OFF position.

- Check all socketed parts for correctly inserted pins. Be sure there aren't any bent under or twisted pins. Check pin 1 locations.

If nothing happens, reapply power and try to feel the components for excessive heat. Be careful as burns may occur if a defective

T A B L E 2 - 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
TM 990/302 (note 1)	4K x 16	2K x 16	OFF	ON	NOTE 2	ON	X	X	X	X
* TM 990/201-41 (note 3)	4K x 16	2K x 16	OFF	OFF	OFF	OFF	OFF	ON	OFF	OFF
* TM 990/201-42 (note 3)	8K x 16	4K x 16	OFF	OFF	OFF	OFF	ON	ON	ON	OFF
* TM 990/201-43 (note 3)	16K x 16	8K x 16	OFF	OFF	OFF	OFF	ON	ON	ON	ON
TM 990/206-41 (note 3)	X	4K x 16	X	X	X	X	ON	OFF	OFF	OFF
TM 990/206-42 (note 3	X	8K x 16	X	X	X	X	OFF	OFF	OFF	ON

X - NOT APPLICABLE
* - Expansion EPROM is not required for Evaluation BASIC. The switch settings to disable all expansion EPROM memory from the system are S1-S4 all OFF. For Development BASIC see paragraph 2.3.5.2.

Note 1: Jumpers E1-E2 and E5-E6 must be installed on the TM 990/302 board when it is used as RAM expansion.

Note 2: If the TM 990/302 is used as the extra EPROM for Development BASIC S3 must be OFF, otherwise it should be ON.

Note 3: Assumes TM 990/302 on board RAM is disabled, if TM 990/302 is used in system.

component is found. If the cause of failure cannot be found, turn power OFF and call your TI distributor. Before calling, please be sure that your power supply, terminal, and all connectors (use a volt-ohmmeter) are working properly.

2.7 RAM EXPANSION

The user may expand the RAM area by configuring additional expansion memory boards into the system. Additional RAM area will provide more user storage area for developing larger POWER BASIC programs. This does not require any modification to the POWER BASIC software package or to the TM 990/101M or TM 990/100M microcomputer boards if configured as described in the preceding sections. When expanding the RAM area, the top of the expansions RAM must be at location $EFFE_{16}$. Recall that the TM 990/302 RAM resides from $E000_{16}$ to $EFFE_{16}$. The expansions RAM must also be continguous from EFFE down. Table 2-7 shows the expansion memory boards currently available as well as the required switch settings to configure each of them to operate with the POWER BASIC configuration as described in Sections 2.3 through 2.3.5.2. Figure 2-10 shows the memory map of the Evaluation BASIC packages on the TM 990/101M and TM 990/100M boards as well as the RAM expansion memory areas. Figure 2-11 shows the memory map of Development BASIC.

2.8 CASSETTE TRANSPORATION OPERATION

POWER BASIC has the capability to "SAVE" and "LOAD" programs from cassette as explained in paragraphs 4.8 and 4.4, respectively. Evaluation BASIC will support only the digital cassette transports of the TI 733 ASR terminal. The following sections and Figure 2-12 will assist the user in operating the cassette units. If more detailed instructions are required, refer to "Model 733 ASR/KSR Operating Instructions," manual number 959227-9701.

Development BASIC can also support audio cassettes if used with the TM 990/302 Software Development Board, for more information than presented here see "TM 990/302 Hardware Reference Manual."

2.8.1 PROCEDURES TO RECORD ON DIGITAL CASSETTES FROM POWER BASIC (SAVE)

Record a POWER BASIC program on cassette as follows:

On the ASR switch panel lower row, verify that the following switches are set as follows:

(1) KEYBOARD to LINE
(2) PLAYBACK to LINE
(3) RECORD to LINE
(4) PRINTER to LINE

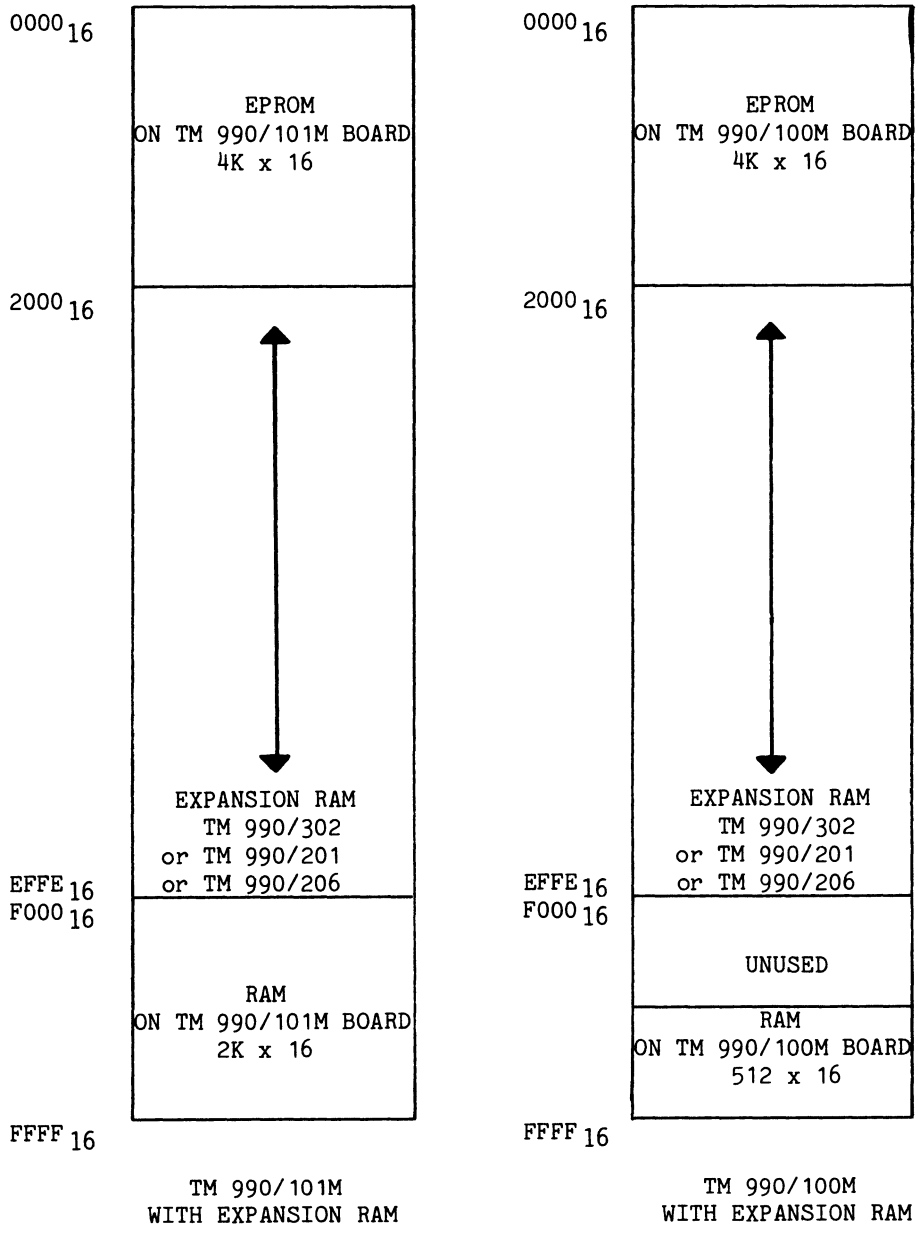


FIGURE 2-10. EVALUATION BASIC MEMORY MAPS

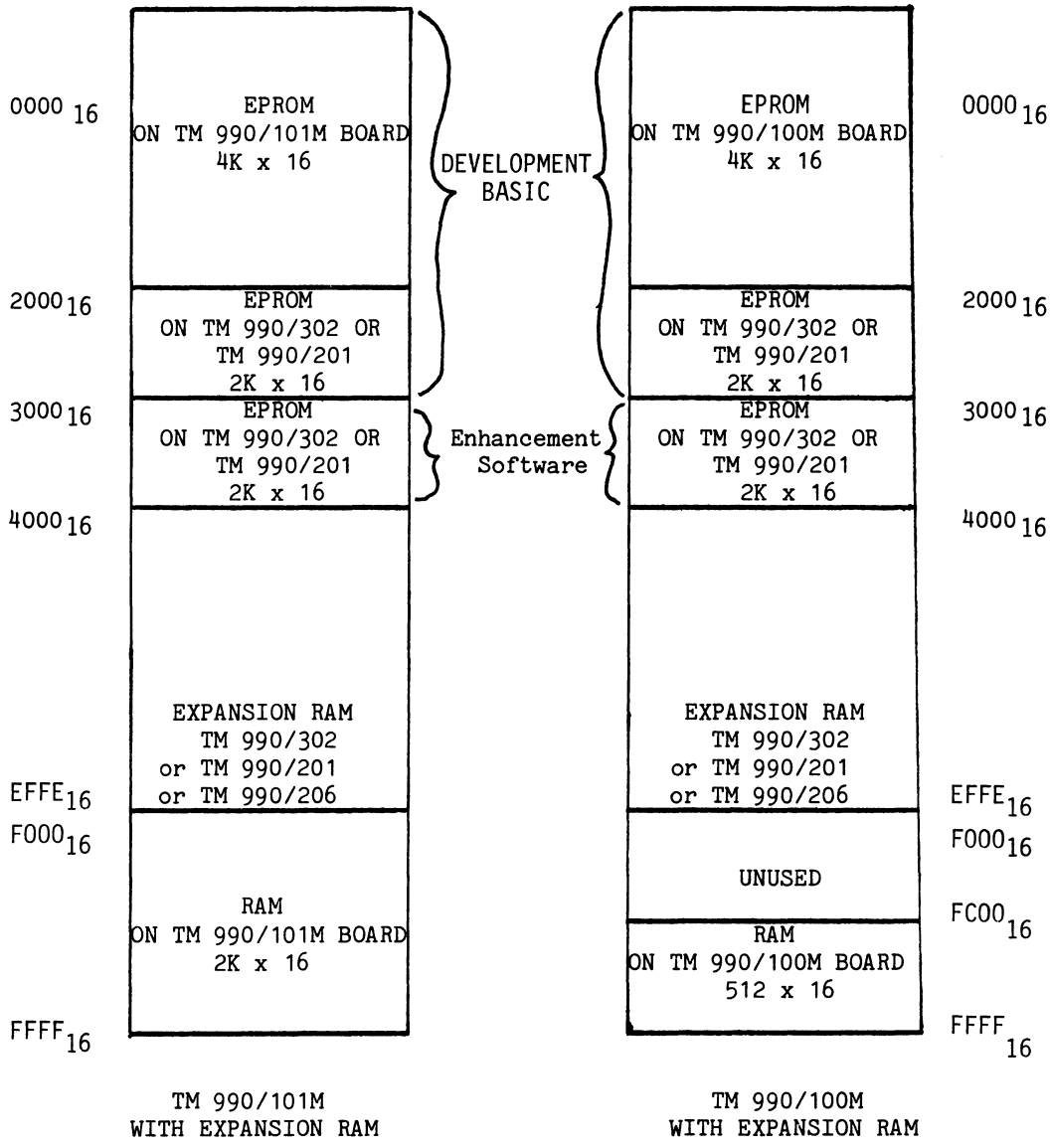


FIGURE 2-11. DEVELOPMENT BASIC MEMORY MAPS

Insert the cassette tape on which the POWER BASIC program is to be "SAVEd" into either cassette transport. Check that the tabs on the bottom of the cassette are not in the write protect position. Then,

(5) Verify terminal is ON-LINE
(6) SET RECORD CONTROL to OFF
(7) Set selected cassette transport to RECORD
(8) Set TAPE FORMAT switch to LINE
(9) Momentarily press REWIND on selected cassette transport
(10) When the END indicator lights, press LOAD/FF on the selected selected cassette transport. The READY indicator lamp should light after a few seconds.

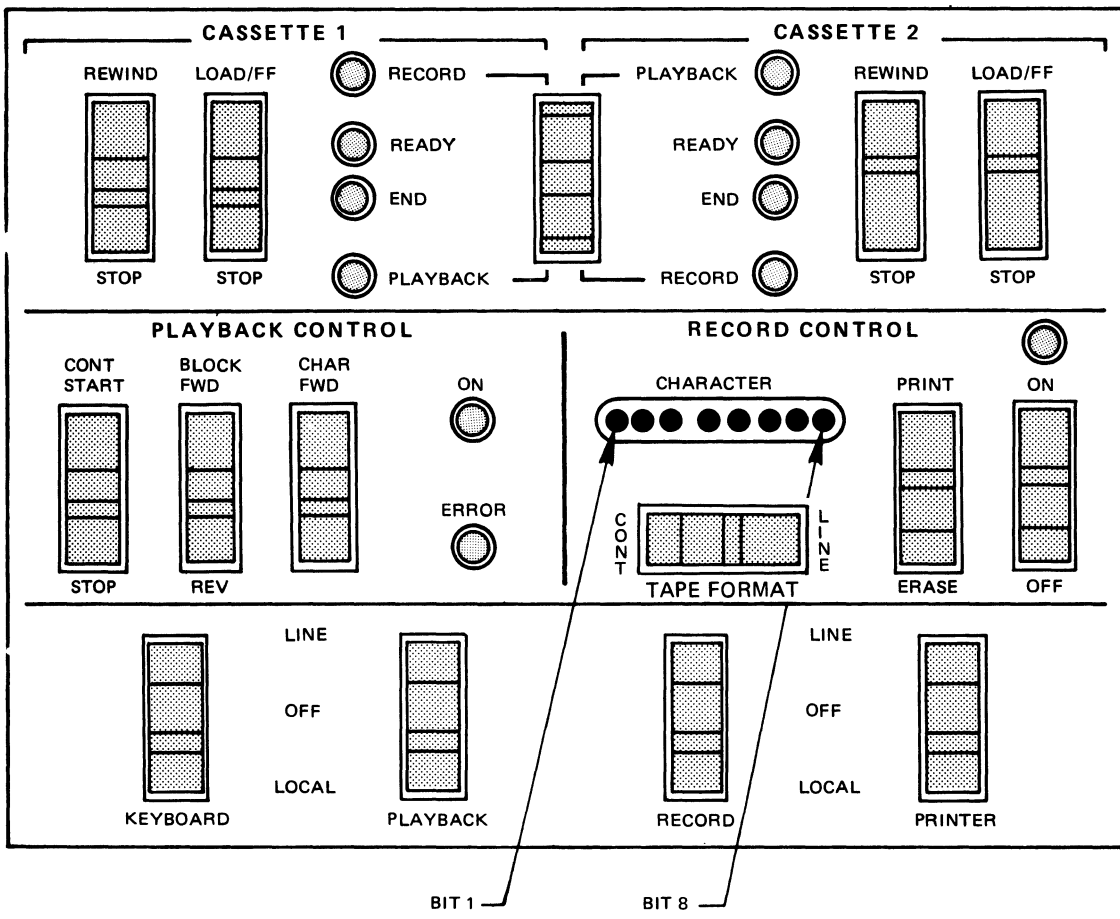


FIGURE 2-12. ASR MODULE ASSEMBLY SWITCH PANEL

(11) If the 733 ASR does not have the Automatic Device Control (ADC) option continue to step 12. With the ADC option, the cassette transport will accept the Record On (DC2) and Record Off (DC4) control characters output by POWER BASIC at the start and end of program saving. If the ADC Option is present, the user may enter the "SAVE" command followed by a carriage return and Evaluation BASIC will automatically turn the record on, start the cassette drive, save the application program, and turn the record off, and stop the cassette. When the cassette stops, RECORD OFF must be pressed to complete the save process.

(12) If the 733 ASR does not have the ADC option, the user must manually start and stop the cassette when "saving". Perform this procedure by readying the cassette transport (step 1 thru 10 above). Enter the "SAVE" command, but do not yet enter the carriage return. Then manually set the RECORD control to ON and enter the carriage return to terminate entry of the "SAVE" command and start the process. The user program currently in POWER BASIC's memory will then be saved on cassette. This continues until the entire program has been recorded. When the cassette stops, the RECORD CONTROL must be pressed to OFF before making any keyboard entries to complete the 'save' process.

2.8.2 PROCEDURES TO RECORD ON AUDIO CASSETTE FROM DEVELOPMENT BASIC (SAVE UNIT #)

Recording on audio cassettes requires:

(1) TM 990/302 Software Development Board
(2) TM 990/452 Enhancement Software Package
(3) Audio cassette recorder

Note, that though only one cassette player is required for SAVEing and LOADing, program development is greatly facilitated by the use of two cassette players: one for playback, and one for record. The procedure to record on audio cassette is:

(1) Insert the appropriate motor control plug into the cassette recorder's motor control jack (the small jack in the "MIC/REMOTE" pair).

(2) Insert the same unit number's record plug into the cassette recorder's AUX jack.

(3) Rewind the cassette to be used (it may be necessary to temporarily remove the motor control plug).

(4) Remove the cassette and rewind the take-up reel, on the right, to position the start of the magnetic tape (not the clear or colored leader) at the edge of the tape head access opening.

(5) Re-insert the cassette into the audio cassette recorder.

(6) Set the cassette for recording (usually by simultaneous depression of RECORD and PLAY buttons; consult your cassette recorder user's manual).

(7) Enter the SAVE command with the appropriate unit number. The cassette unit will record the user's program and stop when loading is complete.

The motor control circuitry can only stop the cassette recorder capstan motor, and not disengage the capstan; if the cassette is not to be used immediately for playback or record the user should disengage the capstan from the pinch roller by pressing the STOP (or equivalent) key. If this is not done a dent can form in the pinch roller causing excessive wow and resulting in cassette errors. To keep the cassette's recording surface clean it is recommended that a cassette by rewound before it is removed.

2.8.3 PROCEDURES TO PLAYBACK TAPE FROM DIGITAL CASSETTES TO POWER BASIC (LOAD)

To load a POWER BASIC program from cassette tape into the POWER BASIC system, proceed as follows:

On the ASR switch panel lower row, verify that the following switches are set:

(1) KEYBOARD to LINE
(2) PLAYBACK to LINE
(3) RECORD to LINE
(4) PRINTER to LINE

Insert the cassette tape containing the POWER BASIC Program to be "LOADed" into either cassette transport. Then,

(5) Verify terminal is ON-LINE
(6) Set selected cassette transport to PLAYBACK
(7) Set TAPE FORMAT to LINE
(8) Momentarily press REWIND on selected cassette transport
(9) When the END indicator lights, press LOAD/FF on the selected cassette transport. The READY indicator lamp should light after a few seconds.
(10) If the 733 ASR does not have the Automatic Device Control (ADC) option, in Evaluation BASIC (or the RDC option in Development BASIC) continue to step 11. With the ADC option the cassette transport will accept the Playback ON (DC1) control character output by Evaluation BASIC to start the cassette and the Playback Off (DC3) at the end of the saved program to stop the cassette. (With the RDC option, the

cassette transport will accept the BLOCK FORWARD commands output by Development BASIC to start the cassette and read the next record.) If the ADC option is present, (an RDC option in Development BASIC), enter the "LOAD" statement followed by a carriage return and POWER BASIC will automatically load and stop the cassette. When the cassette transport stops, press the ESC(ape) key to exit the loading procedure of Evaluation BASIC. Development BASIC automatically returns to the keyboard mode when loading is completed. BASIC then outputs the "*READY" message and the program on cassette is loaded into POWER BASIC and ready for use. If an error occurs during loading, go to Step 12.

(11) If the 733 ASR does not have the ADC option (or RDC Option in Development BASIC), manually start and stop the cassette when "loading". Perform this procedure by reading the cassette transport (steps 1 thru 9 above). Then enter the "LOAD" statement followed by a carriage return and momentarily press the PLAYBACK CONTROL (middle row) CONT START switch; playback transmission and loading will commence. When loading is complete, the ESC(ape) key must be pressed to exit the loading procedure of Evaluation BASIC. Development BASIC automatically returns to the keyboard mode when loading is complete. BASIC will then output the "*READY" message. At this point, the program on cassette will be loaded into BASIC and ready for use. If an error occurs during loading, go to Step 12 below.

(12) When an error occurs, POWER BASIC will print the error number and the statement in error. POWER BASIC will stop loading and return to the keyboard mode when an error is found. Manually stop the cassette transport or subsequent errors will result when attempting to load a program while POWER BASIC is not in the "load mode". To stop the tape transport momentarily press the PLAYBACK CONTROL (middle row) STOP switch. Then attempt to correct the problem to complete loading or re-enter the program into POWER BASIC from the keyboard. Note that all statements on the cassette tape prior to the occurrence of the error will have been successfully loaded and need not be entered again.

NOTE

Evaluation BASIC requires that the 733 ASR terminal operates at 300 baud. Development BASIC allows 1200 baud cassette operations if the 733 ASR is equipped with the Remote Device Control (RDC) option, to respond to BLOCK FORWARD commands. If the 733 ASR does not have the RDC option 1200 baud loading may still be possible, as described in step 11 above, but this will be data dependent. If a cassette will not load at 1200 baud, set the terminal for 300 baud and proceed as in step 11 above.

2.8.4 PROCEDURES TO PLAYBACK AUDIO CASSETTE TO DEVELOPMENT BASIC (LOAD UNIT#)

Reading from audio cassettes requires:

1) TM 990/302 Software Development Board
2) TM 990/452 Enhancement Software Package
3) Audio cassette recorder

The procedure to read from audio cassette is as follows:

(1) Insert either the appropriate motor control plug into the cassette recorder's motor control jack (the small jack in the "MIC/REMOTE" pair).
(2) Insert the same unit number's playback plug into the cassette recorder's earphone or external speaker jack.
(3) Rewind the cassette to be used (it may be necessary to temporarily remove the motor control plug).
(4) Set the cassette for playback, (consult your cassette recorder user's manual).
(5) Enter the LOAD command with the appropriate unit number. The cassette unit will play back the user's program and stop when loading is complete.

All of the cassette handling cautions listed in paragraph 2.8.2 should be observed during playback operations also.

2.9 TWO USER OR SINGLE USER/TWO PARTITION OPERATION

Evaluation BASIC, when used on the TM 990/100M may only be used as a single user/single partition BASIC operation. Evaluation BASIC on the TM 990/101M board may be optionally be configured for single user two-partition operation or for two-user/two-partition operation. Multipartition operation on the TM 990/101 board is performed with the use of the on board 5-switch DIP, the I/O ports P2 and P3, and the main and auxillary TMS9902's.

The five-bit DIP switch is used to:

- Initialize Evaluation BASIC pointers and tables for multiuser/ multipartion operation
- Partition the available user memory between users on a two-user system or between partitions for a single use system.
- Configure both TMS9902 I/O ports on the 101M board

Table 2-8 summarizes the switch settings of the TM 990/101M board.

The ratio represents the memory partition size for (1) each partition of a single user system, or (2) for each user of a two-user system.

Note that Development BASIC does not allow two partition or two user operaiton, and therefore the five-bit DIP switch performs no function in Development BASIC.

2.9.1 SINGLE USER/TWO PARTITION POWER BASIC CONFIGURATION

To configure the POWER BASIC system for single user operation with two user RAM partition areas, the following sequence must be performed:

a) Select the user memory area for each partition by setting the 5 position dip switch as per Table 2-8.

b) Insert board in chassis and connect terminal to connector P2.

c) Apply power to the board and activate the reset switch. This causes POWER BASIC to size RAM from the top ($FFDC_{16}$) down, and divide the available user area between user partitions.

d) Press the "A" or carriage return key on the terminal device. POWER BASIC measures the time of the start bit, determines the baud rate, and initializes the TMS9902 at this baud rate for I/O communications.

e) Evaluation BASIC will print out the banner message and await further keyboard input.

f) Hold down the control key and depress the "T" key to "toggle" control to the other use partition.

g) Repeat steps d) and e) above.

h) At this point both partitions are initialized. Control may be transferred between partitions at any time by using the "(control) T" key as in step f) above. It is no longer required to autobaud the terminal (step d) above. Evaluation BASIC waits for further keyboard entry of commands or statements into either partition of the user's RAM area.

Example:

```
(press RESET switch on microcomputer board)
(strike the "A" key)
TM990 BASIC REV E.x.x
*READY
SIZE
PRGM: 0 BYTES
```

```
VARS: 0 BYTES
FREE: 2000 BYTES
(strike "control T")
(strike the "A" key)
TM990 BASIC REV E.x.x
*READY
SIZE
PRGM: 0 BYTES
VARS: 0 BYTES
SIZE: 1000 BYTES
```

T A B L E 2 - 8
5-BIT DIP SWITCH OPTION

SWITCH POSITIONS					MULTIPARTITION USER AREA	RATIO OF RAM USER AREA #1 TO USER AREA #2
S1	S2	S3	S4	S5		
OFF	OFF	OFF	OFF	OFF	NO	SINGLE USER/ SINGLE PARTITION
OFF	OFF	OFF	OFF	ON	YES	1 : 1
OFF	OFF	OFF	ON	OFF	YES	2 : 1
OFF	OFF	OFF	ON	ON	YES	3 : 1
OFF	ON	OFF	ON	OFF	YES	10 : 1
ON	OFF	ON	ON	OFF	YES	22 : 1
ON	ON	ON	ON	OFF	YES	30 : 1
ON	ON	ON	ON	ON	YES	31 : 1

2.9.2 TWO USER/TWO PARTITION POWER BASIC CONFIGURATION

To configure the POWER BASIC system for two-user operation, the user must perform the following sequence:

a) Select the user memory area for each partition by setting the 5-switch dip as per Table 2-8.

b) Insert board into chassis and connect the two terminals to connectors P2 and P3 on board.

c) Apply power to the board and activate the RESET switch. This causes Evaluation BASIC to size RAM and divide the available user RAM area between the two users.

d) Press the "A" or carriage return key on either terminal device. POWER BASIC measures the time of the start bit, determines the baud rate, and initializes the TMS9902 I/O port connected to this terminal at this baud rate.

e) Evaluation BASIC will print out the banner message on this terminal and await further keyboard input.

f) Press the "A" or carriage return key on the other terminal device. POWER BASIC measures the baud rate and initializes the second TMS9902 at this baud rate for this terminal. Note that the two terminal devices do not need to be initialized to the same baud rate.

g) Evaluation BASIC will print out the banner message on this terminal and await further keyboard input.

h) At this point both user program areas are initialized, and each user may proceed to enter his sequence of POWER BASIC statements and commands. At any point either user may "swap" his entire user partition (including program statements, variables, etc.) with the other user partition by striking the "(control) T" key. This operation will be performed even if one or both of the user programs are in execution.

Example:

```
(press RESET switch on microcomputer board)
(strike the "A" key on one of the terminals)
TM990 BASIC REV E.x.x
*READY
SIZE
PRGM:   0 BYTES
VARS:   0 BYTES
FREE: 2000 BYTES
```

```
(on the other terminal)
(strike the "A" key)
TM990 BASIC REV E.x.x
*READY
SIZE
PRGM:    0 BYTES
VARS:    0 BYTES
FREE: 1000 BYTES
```

2.9.3 COMMUNICATIONS BETWEEN MULTIPARTITIONS

When using either the two-user/two partition or single user/two partition option, programs can communicate with each other through a set of common dimensioned variables: COM(0) through COM(9). These variables are predefined variables common to each partition and do not require explicit dimensioning. The linkage between two programs in the two partitions is not established until the "RUN" command has been executed by both partitions. The "NEW" command terminates the link between the two user partitions and also clears all program and variable storage associated with the partition in which the "NEW" command was executed.

2.9.4 EXAMPLE

Typically, one of the "common" variables is used for control of communication while the others are used for data transfer. The following example illustrates their usage.

PARTITION #1

```
10  DIM A(8)
20  COM(0)=0
30  INPUT $A(0)
40  IF COM(0)<>0 THEN GOTO 40
50  $COM(1)=$A(0)
60  COM(0)=1
70  GOTO 30
```

PARTITION #2

```
10      IF COM(0)=0 THEN GOTO 10
20      PRINT $COM(1)
30      COM(0)=0
40      GOTO 10
```

Partition #1 inputs characters into array $A(0) and sends the string to partition #2 when COM(0) has been cleared. Partition #2 waits for the message by monitoring COM(0) is reset or readied for another message.

2.10 EPROM PROGRAMMING

Development BASIC has the facilities to program user software into TMS2716 EPROMs. The programming of EPROM's requires:

1) TM 990/302 Software Development Board
2) TM 990/452 Enhancement Software Package
3) TM 990/451 TMS2708/TMS2716 Personality Board
4) A power supply with +35V to +55VDC output

CAUTION

Power connections and Personality Board installation should never be attempted with system power on.

The procedure to program TMS2716 EPROM's is:

(1) Turn off system power
(2) Connect the 35V-55V power supply to the terminal block-TB1 on the TM 990/302 board (see paragraph 2.3.1).
(3) Install the TM 990/514 EPROM personality board on the 50 pin edge connector-P3 on the TM 990/302 board; the TM 990/514 should be positioned so that the stake pin jumpers are on the right side.
(4) Position all three of the stake pin jumpers on the TM 990/514 to select "2716" EPROM's.
(5) Turn system power on.
(6) Enter (or LOAD) the user program to be programmed into EPROM
(7) Insert a TMS2716 into the socket on the TM 990/514; pin one is the upper right corner of the socket.
(8) Program the EPROM (see paragraph 4.6)
(9) Repeat steps 7 and 8 as necessary.

2.11 LOAD VECTOR

Development BASIC can be "warm started" (see paragraph 3.12) by use of the LOAD trap vector. The simplest way to initiate the LOAD is via a normally open push-button switch connected between backplane (P1) pin 93, and ground. On the TM 990/510, and the TM 990/520 Card Chassis P1 pin 93 is available on the backplane terminal block, and is labeled RESTART.B; this will be the simplest place to connect a push-button. A 39 uf capacitor must be installed on the CPU card to act as a debouncing circuit for the LOAD pushbutton: on the TM 990/100M this is C5, on the TM 990/101M this is C18.

2.12 TM 990/101M SECOND EIA PORT

In Development BASIC the second EIA port on the TM 990/101M CPU board can be connected to an EIA compatible output device; such as a TI Model 810 line printer. The second port is only written to, never

read from. The baud rate of the second port is initialized via the BAUD statement (see paragraph 5.8.4) and output to it is controlled via the UNIT statement (see paragraph 5.8.3). Baud rates available and connection techniques are the same as for the main port (see paragraphs 2.2.5 and 2.3.6.2).

FUNCTIONAL DESCRIPTION

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. Commands begin with the command name (or the first three letters of the command name) 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>

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

<line number><statement 1>::<statement 2>:: ...

The last statement on a line must be totally contained on that line (it cannot be continued on the next line), and terminated with a carriage return character.

POWER BASIC will generate automatic line number prompts for the user, to facilitate simple program statement entry. Auto-line numbering is initialized to begin at statement number 10 and generates an increment value of 10 between subsequent statement numbers.

To initiate auto-line numbering when generating a program, 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). To terminate auto-line numbering, enter a carriage return at the end of the statement.

POWER BASIC programs are executed beginning with the lowest numbered line and proceeding with the next 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 also is used to associate program editing activities with a particular statement line in the program.

3.4 SOURCE STATEMENT FORMAT

3.4.1 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>". Note that Evaluation BASIC does not support direct output of non-printable ASCII control characters.

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.5 EDIT MODE COMMANDS

To aid in program writing and debugging, an advanced editor is contained in POWER BASIC. The editor uses the following special control characters:

CR	Enter edited line
(ln)(ctrl)E	Display line for editing
(ctrl)F	Forward space cursor
(ctrl)H	Backspace cursor
SPACE	Space or remove character
RUBOUT	Backspace and remove character
(ctrl)Dn	Delete n characters*
(ctrl)In	Insert n blanks*

*Not supported by Evaluation BASIC

The phrase "(ctrl)" indicates that the user holds down the control key while depressing the key corresponding to the character immdediately following. For example:

"(ctrl)H"

means depressing the "H" key while holding down the key marked "CTRL" or "CTL". The character is not echoed on the terminal nor is it stored in the input buffer. All illegal control characters are echoed as a bell and otherwise ignored.

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

An additional feature allows editing program lines that have previously been entered. The form is:

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

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

The following examples illustrate the character insertion and deletion features of Development BASIC. Editing features, "(ctrl)Dn" and "(ctrl)In", are not supported by Evaluation BASIC. The cursor position is designated by "_".

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.

Evaluation BASIC: The (ctrl)In and (ctrl)Dn features are not supported by Evaluation BASIC.

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.

Evaluation BASIC: Note that hexadecimal constants are not supported by Evaluation BASIC.

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 Development BASIC, or 7 significant digits in Evaluation BASIC, and may not be larger than 10^{75} or have a negative exponent less than 10^{-75}. Real numbers may be expressed simply as a number followed by a decimal fraction, or may also have an 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 vice-versa. (Example: 'THIS IS A STRING', "SO'S THIS".) 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 out 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' + %39 :: $TST = $TST + "T"
```

will place the desired five character string into the variable $TST. The % operator enables the Evaluation BASIC user to insert nonprintable ASCII character codes into string constants for output by inserting the decimal ASCII code following the % symbol into the character string. For additial 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).

Evaluation BASIC: Note that direct output of ASCII characters is not supported by Evaluation BASIC.

3.7 VARIABLES

POWER BASIC supports 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.

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, AO, A123.

Invalid names: ABS (function name), A.B (non-letter), 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 apply to numeric array variables with the additional specification that numeric array variables must appear in a DIM statement which is executed before the first reference to the variable (see DIM statement, Section V, paragraph 5.3). Numeric array variables must always appear with a subscript. The subscript distinguishes an array 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 Evaluation BASIC is composed of 32 bits, or 4 eight-bit bytes. Thus, a maximum of three characters should be stored in a simple string variable of Evaluation BASIC, 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 in overwriting of data immediately following the string variable. Note that Development BASIC supports simple variables which are 48 bits or six 8-bit bytes in length. Simple variables in Development BASIC can therefore contain a maximum of five characters terminated by a null.

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.

The 48-bit Development BASIC 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. 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.

Note that the following example on the 32-bit Evaluation BASIC functions in the same manner as the above example, except that the elements are smaller (4 bytes maximum per array element).

```
            DIM A(2,2)
$A(0,0)  :  RHIN(OCEROR)  $A(0,1) : OCER (OS)  $A(0,2) : OS
$A(1,0)  :  ELEP(HANT)    $A(1,1)   HANT       $A(1,2) :
$A(2,0)  :  GIRA(FFE)     $A(2,1)   FFE        $A(2,2) :
```

3.7.5 RESERVED VARIABLES

Evaluation BASIC supports a multi-partition user program area. Both terminals may access their separate partition or both partitions may be assigned to a single terminal. In order to facilitate communication between these partitions, a set of common variables has been designated. These variables are contained in an array named COM consisting of ten elements (0-9) and must be referenced as a dimensioned variable (COM(0)-COM(9)). These variables are reserved and should be used for inter-partition communications only. These command variables differ from other variables used in BASIC application programs because the common variables are not deleted when the "NEW" statement is executed as are all other user defined variables. For additional details on the multi-partition feature of Evaluation BASIC, refer to Section II, paragraph 2.9 . Note that Development BASIC does not support a multi-partition user program area, therefore the variables COM(0) through COM(9) are not reserved variables of Development BASIC and are not predefined array variables. Caution should be used in generating POWER BASIC programs on Development BASIC using variables COM(0) through COM(9) and then transporting these programs to Evaluation BASIC where COM(0) through COM(9) are used to perform additional functions.

3.7.6 VARIABLE STORAGE

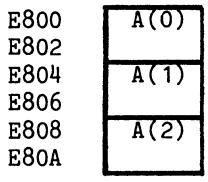
The following paragraphs will explain the internal variable storage structure used by POWER BASIC. This will be helpful when accessing varibles of BASIC from a "CALLed" assembly languge subroutine.

3.7.6.1 NUMBER ARRAY STORAGE

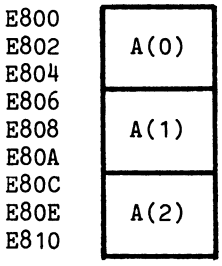
Arrays of numbers are stored in memory by now with each number (element) occupying 4 bytes in Evaluation BASIC and 6 bytes in Development BASIC. The storage of singly and doubly dimensioned arrays are illustrated in the two diagrams below. Larger dimensioned

arrays are stored in a similar manner.

Single dimensioned array A with 3 elements starting at Hex address $E800_{16}$:

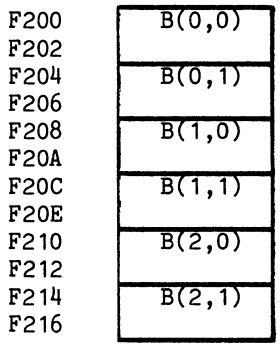


EVALUATION BASIC

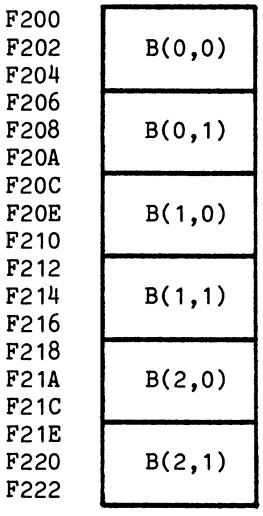


DEVELOPMENT BASIC

Doubly dimensioned array B with 3 rows (first subscript) and 2 columns (second subscript) starting at hex address $F200_{16}$:



EVALUATION BASIC



DEVELOPMENT BASIC

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

```
ARRAY BASE + 4 * (SUBSCRIPT)      EVALUATION BASIC
ARRAY BASE + 6 * (SUBSCRIPT)      DEVELOPMENT BASIC
```

eg. A(1) above would be:

```
E800 + 4 * 1 = E804               EVALUATION BASIC
E800 + 6 * 1 = E806               DEVELOPMENT BASIC
```

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

```
ARRAY BASE + 4 * (MULTIPLER*SUSCRIPT1 + SUBSCRIPT2)  EVALUATION BASIC
ARRAY BASE + 6 * (MULTIPLER*SUBCRIPT1 + SUBSCRIPT2)  DEVELOPMENT BASIC
```

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

```
F200 + 4 * (2*1+0) = F208         EVALUATION BASIC
F200 = 6 * (2*1+0) = F20C         DEVELOPMENT BASIC
```

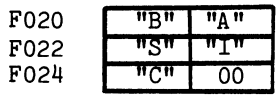
3.7.6 2 STRINGS AND STRING ARRAY STORAGE

Strings are stored one ASCII character per byte, and are terminated with a null byte. Evaluation BASIC variables are 4 bytes in length, while Development BASIC variables are 6 bytes in length. The examples below show the string storage format.

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

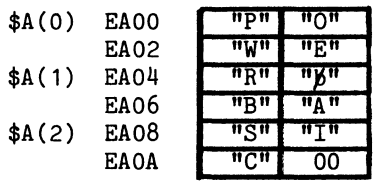
EVALUATION BASIC DEVELOPMENT BASIC

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

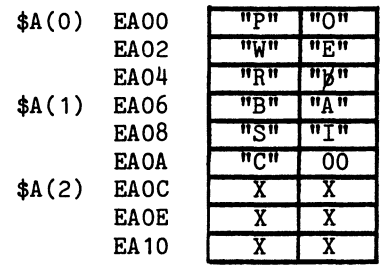


Strings may be empty or they may have any length up to their declared maximum. Care must be taken that strings of lengths larger than specified maximum are not placed into simple string variables, or other variables may be written over.

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 $EA00_{16}$.



EVALUATION BASIC



DEVELOPMENT BASIC

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

"PRINT $A(0), $A(1), $A(2)" in Evaluation BASIC will result in

POWER BASIC R BASIC SIC

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

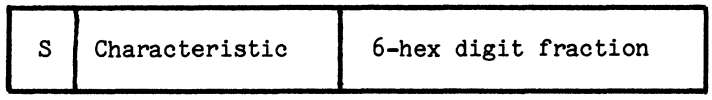
POWER BASIC BASIC

3.7.7 VARIABLE FORMAT AND ACCURACY

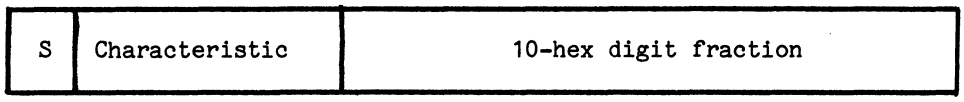
Any variable may contain an ASCII character string, a number, or both. Variable contents are completely program context dependent. Floating-point quantities in POWER BASIC are represented in either 32 or 48 bits, and are termed short and long, respectively. Evaluation BASIC utilizes the 32-bit representation, while Development BASIC uses the 48-bit representation. In either case, 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, either in positions 8-31 in Evaluation BASIC or in positions 8-47 in Development BASIC, 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}). Evaluation BASIC utilizing 32-bit floating point representations provides approximately 7 digits of accuracy, while the 48-bit Development BASIC provides approximately 11 digits of accuracy.

Evaluation BASIC Floating-Point Format:

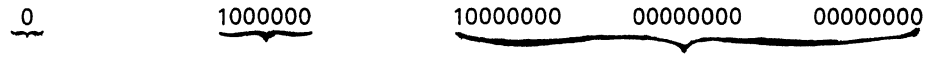


Development BASIC Floating-Point Format:



Examples:

The pattern



sign characteristic fraction

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 deontes a positive number, the number represented is $+0.5*16^{0} = 0.50$.
The pattern

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 zeroes in bit positions 0 through 15 followed by the two's complement 16-bit integer in bit positions 16 through 31.

Evaluation & Development BASIC Integer Format:

00000000 00000000	two's complement integer

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 floating-point format.

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. Note that the BIT function is not supported by Evaluation BASIC.

```
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 00000000000000000000000100000000000000000000
? 1.5           0 1000001 00011000000000000000000000000000000000000000
? 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.

Evaluation BASIC: Note that Evaluation BASIC supports exponentiation to integer powers only. If a non-integer exponent is used, the fractional portion is truncated, and the value is raised to the integer power.

Development BASIC: Note that Development BASIC supports exponentiation to any floating point quantity. Both positive and negative exponents are valid. However, since Development BASIC uses logarithms to calculate the exponentiation, only positive quantities may be raised to a given power.

3.8.2 ARITHMETIC EXPRESSIONS

An arithmetic expression is any valid sequence of numbers, variables, operators, and parentheses (valid meaning that parentheses must be properly balanced, no two numbers or variables can be adjacent, 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 be a 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
```

Evaluation BASIC: The logical operators are not supported by Evaluation BASIC.

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 = OAAAAH (hex "AAAA"), and B=05555H (hex "5555") and C = OBBBBH, (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.

Evaluation BASIC: The approximately equal (==) and the boolean operators NOT, AND and OR are not supported by Evaluation BASIC.

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. *,1/
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:

Example:

```
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

POWER BASIC 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 its execution. This line may contain multiple statements properly separated by a double colons.

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 and returns to keyboard mode after any error, STOP, or when all statements have been executed.

The following examples illustrate one 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, but 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	PROGRAM
LIST	RUN
LOAD	SAVE
NEW	SIZE

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 code numbers corresponding to the appropriate error message to indicate which error has occurred. This is the case for the TM 990/450 Evaluation BASIC as well as the TM 990/451 Development BASIC. However, Development BASIC utilizing the TM 990/452 Development BASIC Enhancement EPROM set provides the capability of displaying the error message itself rather than the error code number for all errors generated by the Development BASIC package. Note that if this expansion EPROM set is not present in the

Development BASIC system, all errors will be displayed as an error code number only.

POWER BASIC reports all errors using basically two formats. The first format displays the error code of error message and the statement number where the error occurred according to the following format(s):

```
*ERROR XX AT YYYY
XXXXXXXXX AT YYYY
```

where:

XX is the error code or error message
YYYY is the statement number

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 code or 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 cassette. They indicate that the most recently entered statement or command, on 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 satement 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. If an error other than syntax occurs during command or keyboard statement execution, only the error is output. Any syntax errors may then either be corrected and the statement or command executed again, or the LOADing operation may be continued or repeated (see the LOAD command of Section 4, paragraph 4.4).

The following error codes and error messages may be issued by the POWER BASIC package:

CODE ERROR MESSAGE

 1 = SYNTAX ERROR
 2 = UNMATCHED PARENTHESIS
 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 (<#D<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 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 = EXPANSION EPROM NOT INSTALLED
42 = INTERRUPT W/O TRAP
43 = INVALID BAUD RATE
44 = TAPE READ ERROR
45 = EPROM VERIFY ERROR
46 = INVALID DEVICE NUMBER

3.12 RESET AND LOAD FUNCTION OPERATION

The RESET function is used to initiate the Power-up initialization sequence of POWER BASIC as explained in Section 2, paragraph 2.4.2. The RESET function may also be used at any time during the execution of the POWER BASIC application program. The RESET function is actuated by activating the RESET switch near the corner of the TM 990/100M or TM990/101M microcomputer board. The RESET function performs the following operations in POWER BASIC.

The RESET function automatically sizes and clears the systems RAM area starting from memory location $FFDC_{16}$ and checks sequential memory locations down through memory location 4000_{16} until a write/read mismatch is detected. (Note that POWER BASIC detects the resulting "hole" in the memory map when a TM 990/100M microcomputer board is being used and continues sizing and clearing memory from $F000_{16}$ on down.) All of the detected RAM area is then allocated to POWER BASIC to be used for system overhead and user program area. The lower bounds of RAM may later be changed by using the optional address parameter of the NEW command of Development BASIC (see Section 4, Paragraph 4.5).

POWER BASIC then performs the auto-baud sequence to initialize the serial I/O interface from terminal communications. POWER BASIC waits for the user to enter the "A" key (or carriage return) on the terminal device and then measures the time of the start bit, determines the baud rate, and initializes the TMS9902 to this baud rate.

Next it initializes all POWER BASIC pointers at their beginning, essentially performing the NEW command.

In Development BASIC, the initialization sequence then restores the UNIT flag to a value of 1, that is, it directs all output to Port A on the microcomputer board (for additional information refer to the UNIT statement of Section 5, paragraph 5.8.3).

POWER BASIC then outputs the following message:

```
TM 990 BASIC REV  X.n.m
*READY
```

where:

X = language level
n = the release number
m = the revision number

POWER BASIC is then intialized and awaits user keyboard input.

Therefore, it can be seen that restarting POWER BASIC by using the RESET switch results in the user's BASIC program being destroyed - an unacceptable consequence in many situations. For this reason, Development BASIC provides the capability of performing a "warm start" procedure which will not destroy the user's program. This is performed by using the LOAD function of the TM 990/100M or TM 990/101M microcomputer boards.

The LOAD function is actuated by setting the RESTART.B signal on connector P1 to a logic ZERO, or more commonly by activating a push button switch configured between the signal RESTART.P and GROUND on the TM 990/510 card chassis. When the LOAD function is activated it causes a trap to the WP and PC values at memory locations $FFFC_{16}$ and $FFFE_{16}$, respectively. The LOAD function performs the following operations in Development BASIC.

The LOAD function restores the UNIT flag of Development BASIC to a value of 1, that is, it directs all output to Port A of the microcomputer board. (For additional information refer to the UNIT statement of (Section 5, paragraph 5.8.3).

Next, POWER BASIC outputs the banner message:

```
TM 990 BASIC REV  X.n.m
*READY
```

where:

X = language level
n = the release number
m = the revision number

POWER BASIC then awaits user keyboard inputs to edit, list, or execute the user's BASIC application program.

POWER BASIC REFERENCE GUIDE

SECTION IV

BASIC COMMANDS

4.1 GENERAL

POWER BASIC programs are created, executed, and debugged through interaction with the BASIC system. The system recognizes two kinds of input: statements and commands. BASIC commands direct and control system functions which include initiating computer operation, storing data, and listing 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 in the form of a keyword which may be abbreviated to the first three letters. For example:

```
LIST
```

can be entered as

```
LIS
```

4.2 CONTINUE COMMAND

Form:

```
CONtinue
```

The CONTINUE command transfers control to the next statement of the BASIC program. (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.

Evaluation BASIC: The CONTINUE command is not supported by Evaluation BASIC.

4.3 LIST COMMAND

Forms:

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

The LIST command displays all or any portion of the current program. Entering only the command forces the entire program to be listed. By entering a line number, specific portions of the program can be listed. The line number specifies the starting line number where listing of the program is to begin. The starting line number need not 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 either the last line number of the program or when the user enters the ESCape key.

Example:

```
LIST
```

results in a listing of an entire program, while

```
100 LIST
```

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

4.4 LOAD COMMAND

When a user program has been properly "SAVEd" on cassette, the LOAD command loads the user program from cassette into memory.

Forms:

```
LOAD
LOAD <exp>
```

Development BASIC supports both forms of the LOAD command, while Evaluation BASIC supports only the first form.

Loading a BASIC program into memory inserts only those BASIC statements with statement numbers on the cassette and will not affect BASIC statements already in memory having different statement numbers. Any statements in memory which have the same statement number as the program on cassette will be overwritten when the cassette is loaded.

The LOAD command without an expression, or with an expression value of zero, will result in the user's BASIC program being loaded from the 733 ASR digital cassettes. To load a program from digital cassette, the transport must be readied and in the playback mode before the LOAD

command is executed. Reference Section 2, paragraph 2.8 for details on 733 ASR cassette transport loading and operation. Should any error occur during program loading from 733 digital cassettes, the offending statement will be printed or the terminal device along with the appropriate error messages and the statement number where the error occurred. When an error occurs, the loading procedure is terminated and POWER BASIC returns to the keyboard mode. The user must then manually stop the cassette transport in Evaluation BASIC or subsequent errors will result when attempting to load a program while POWER BASIC is not in the "load mode". Development BASIC automatically stops the cassette transport on an error. Note that all statements on the cassette tape prior to the occurrence of the error will have been successfully loaded and need not be entered again. The remainder of the statements on the cassette may be loaded by again entering the "LOAD" command after manually stopping the cassette transport. When loading is complete, the 733 ASR cassette drive stops and the user must strike the ESCape key on the keyboard to return to the keyboard mode in Evalution BASIC. When loading of a SAVED digital tape is complete in Development BASIC, it automatically returns to the keyboard mode and awaits command/statement entry. Program listing, editing, and execution may then proceed.

The LOAD command with an expression value of 1 or 2 will load the program from audio cassette drive # 1 or #2, respectively. To load a program from audio cassette, the tape drive must be readied and in the playback mode before the LOAD command is executed. Reference Section 2, paragraph 2.8.2 for details on audio cassette drive setup and operation. Note that the audio cassette device service routines of POWER BASIC reside in the TM 990/452 Development BASIC Enhancement Software Package as presented in Section 1, paragraph 1.3. Attempts to execute the "LOAD 1" or "LOAD 2" commands without the Development BASIC Enhancement EPROM set installed in the system will result in an ERROR 41 (EXPANSION EPROM NOT INSTALLED).

NOTE

The audio cassette device service routines cannot be interrupted during loading of a program since each bit of the data bytes has a specified minimum and maximum pulse width for reliable data storage and retrieval. Therefore, all interrupts are masked at the CPU whenever a LOAD is being performed to device 1 or 2. This implies that the real-time clock of POWER BASIC will not be updated for the entire LOAD process. This time period can accumulate to a significant amount. Therefore, the real-time clock is stopped and zeroed when the LOAD process from audio cassette is begun to emphasize the resulting clock innaccuracy.

When loading from audio cassette, the "PROGRAM ENABLE" LED on the TM 990/302 board is turned on and off at the beginning and end of each record as it is read by Development BASIC.

If a tape read error or checksum error occurs during loading from audio cassette, the loading procedure is terminated and POWER BASIC returns to the keyboard mode, stopping the audio casette drive. If an error occurs during translation of the "loaded" statement, the offending statement will be printed on the terminal device along with the appropriate error message and the statement number where the error occurred. The loading procedure will then be terminated and the audio cassette drive must manually be stopped by the user. Note that all statements on the cassette tape prior to the occurrence of the error will have been successfully loaded and need not be entered again. If an error occurs the user may attempt to re-read the entire audio cassette or read the remainder of the cassette. When loading is complete, the audio cassette drive stops and POWER BASIC returns to the keyboard mode for command/statement entry.

At any time during program loading from either audio on digital cassettes, the user may terminate the loading process by entering the ESCape key. In Evaulation BASIC, the 733 ASR digital cassettes must be stopped manually by the user. In Development Basic, the auido cassette drive and the 733 ASR digital cassette will automatically be stopped by POWER BASIC. POWR BASIC will then return to the keyboard mode for command/statement entry.

All statements which were read before the entry of the ESCape key will successfully be loaded into POWER BASIC.

The LOAD command with an expression value greater than 2 will result in the expression being interpreted as a memory address. Typically, the expression is entered as a hexadecimal memory address constant. The form of the LOAD command with an address parameter is used by Development BASIC to initialize the internal pointers of POWER BASIC to reference a BASIC application program residing in EPROM at the specified memory address. Also the pointers to the user RAM area are set to the lower RAM address detected by BASIC or as set by the "NEW address " command, to provide a larger area for BASIC application program variable and array storage. The BASIC application program would previously have been stored into EPROM using the PROGRAM command at the specified memory address corresponding to the starting address at which these EPROM's were placed in the memory map of Development BASIC. After execution of the "LOAD address" command, the user may list and execute the BASIC program referenced by the address. Note that the user may not perform any program editing of this program since it resides in EPROM and cannot be modified once stored in EPROM. If editing of a BASIC program in EPROM is attempted, unpredictable results may occur. Also the user must verify that the specified address is a valid EPROM application starting address or

unpredictable results will occur. The user may execute several "LOAD <address>" commands to sequentially access multiple EPROM application programs, however only the most recently "loaded" EPROM application is accessable and executable by POWER BASIC.

The user may return to the development mode of POWER BASIC which permitting new program entry, editing, and debug by entering the NEW command. Reference Section 4, paragraph 4.5 for additional information of the NEW command.

Examples:

```
LOAD              (LOAD program from 733 ASR digital cassette)
LOAD 0            (LOAD program from 733 ASR digital cassette)
LOAD 2            (LOAD program from audio cassette drive #2)
LOAD 05000H       (Initialize pointers to application program
                    in EPROM residing at hex address 5000)
```

Evaluation BASIC: The LOAD command with an expression or memory address parameter is not supported by Evaluation BASIC.

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 Development BASIC to limit the amount of RAM memory which can be used by the POWER BASIC system for interpreter overhead and user program area. When POWER BASIC is initialized during power-up, it automatically sizes and clears the system RAM area starting from memory location $FFDC_{16}$ checking sequential memory locations down through memory location 4000_{16} 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 the workspace registers of all BASIC callable assembly language subroutines and all assembly language interrupt handlers. For this reason, the NEW command with the address parameter was introduced. The lower bound of RAM memory is set to the specified address, and all

POWER BASIC pointers are initialized to correspond to this new memory configuration. Therefore an area of RAM can be reserved for application use from the specified address on down toward memory address 4000_{16} . Care must be taken when specifying the address, to be sure that RAM actually does exist at the specified address and that RAM is contiguous from that address up to high memory, or unpredictable results may occur. The user must also be sure that the software which uses the free area does not overlap past the specified address. In addition to setting the lower memory bound, the NEW command also deletes the current user program, and clears all variable space, pointers, and stacks. (Note that the PROGRAM command of Development BASIC uses an additional 8 bytes below the specified lower bound RAM memory address for pointer storage during EPROM programming.)

Examples:

```
NEW
NEW  0E800H
```

Evaluation BASIC: The form "NEW <address>" is not supported by Evaluation BASIC.

4.6 PROGRAM COMMAND

Form:

PROgram

The PROGRAM command is used to activate the EPROM Programmer Package of Development BASIC. This package is used to program the user's application into TMS2716 EPROM's for future execution. The PROGRAM command is not supported by Evaluation BASIC. Also note that the PROGRAM command resides in the TM 990/452 Development BASIC Software Package Enhancement EPROM set as presented in Section 1.1.3. Attempts to execute the PROGRAM command without the Development BASIC Enhancement EPROM set being installed in the system will result in an ERROR 41 ("EXPANSION EPROM NOT INSTALLED") error.

The PROGRAM command programs the entire POWER BASIC application program currently in user memory into the TMS2716 EPROM; it programs all internal application program pseudo-code, including variable definition table, statement location table, and internal statement pseudo-code. It does <u>not</u> program the "source" code as input by the user or as displayed by the LIST command.

The EPROM set programmed with the user's application is completely relocatable, and may be placed anywhere within the vacant address space of Development BASIC.

After programming the application into EPROM, the "LOAD <address>" command will initialize BASIC to permit user access to the program in EPROM. The user may list and execute the program in EPROM, however, attempts to edit the program will result in erratic operation.

When programming the BASIC application program into EPROM, the user has the option to enable or disable immediate application program execution after system power-up and initialization. The power-up and execute feature is stored as the first word of the EPROM set being programmed with the users application. In addition, the EPROM set must be placed at address location 3000_{16} if the option is to be used. Note that the TM 990/452 Development BASIC Enhancement Software Package EPROM set also resides at location 3000_{16} and must be removed if the power-up option is to be used. If the power-up option is not to be used, the application EPROM's may be placed at any other vacant location within the memory map of Development BASIC. The power-up and execute word at location 3000_{16} is tested whenever the hardware RESET switch of the microcomputer board is operated. If the correct bit pattern is read from location 3000_{16}, the user's BASIC application program residing at 3000_{16} will immediately begin execution. If the correct bit pattern is not detected, Development BASIC will be initialized to the keyboard mode, display the banner message, and await user keyboard input.

The SIZE command will display the "PRGM" size of the current user application program including overhead. This indicates the number of bytes that will be programmed into the EPROM set when the PROGRAM command is issued. If the user has a large application program which is to be stored into EPROM, and there is concern about the large size of the program, the user may remove all REMark statements and tail remarks(!) from the application program. Removing even a few remark entries will result in an appreciable reduction in the program size since each character of a remark entry consumes a byte of program storage. Also, many unnecessary variables may be stored in the symbol table of POWER BASIC during application program development. These variables have no meaning in the application program. For this reason, it is recommended that the user perform a SAVE, NEW, LOAD command sequence prior to programming the application into EPROM. The SAVE command will detranslate and store the application program on either digital or audio cassette (the extraneous variables are not saved on the cassette). The NEW command will then delete the user application program, and clear all variable space, pointers, and stacks. The LOAD command will retrieve the application program from cassette. All extraneous variables will have been removed from the symbol table, and the application program will be ready to be programmed into EPROM's.

Prior to entering the PROGRAM command, the user should verify that the TM 990/302 Software Development Board and the TM 990/514 EPROM Personality Module are properly configured for TMS2716 programming. Refer to Section 2, paragraph 2.10 for detailed information on the set up of the Software Development Board and EPROM Personality Module, as well as Programming power supply configuration.

When the PROGRAM command is entered, POWER BASIC responds with the following prompt and awaits user input.

```
RUN? _
```

The "RUN" parameter specifies whether activation of the RESET switch of the microcomputer board will a) initialize Development BASIC to the keyboard mode awaiting user keyboard input, or b) enable immediate execution of the application program in EPROM. A "Y" response will enable immediate execution of the application program in EPROM upon power-up, a "N" response will result in POWER BASIC being initialized to the keyboard mode upon activation of the RESET switch, and an ESCape (or BREAK) key response will exit the EPROM programmer and return control to the POWER BASIC executive. Any other response will result in the prompt being reissued. (Recall that to utilize the automatic power-up and run option, the application EPROM's must reside at memory address 3000_{16} in the final system in addition to a positive response to the "RUN?" prompt.

Upon entry of a valid response, POWER BASIC will respond with the following prompt and await user input.

```
MOUNT EPROM
EPROM READY? _
```

The user should verify that the EPROM module is properly configured and that the TMS2716 EPROM is correctly installed. The user should enter a "Y" response when the EPROM is ready to be programmed. If the user enters the ESCape (or BREAK) key, Development BASIC will exit the EPROM programmer and return to the keybaord mode, awaiting BASIC command/statement input. Any response other than the "ESCape" or "Y" entry will result in the prompt being reissued.

Upon entry of the "Y" response, the EPROM personality card is verified as being correct for use in TMS2716 programming. If the personality card is of incorrect type or is configured incorrectly, the "EPROM READY?" prompt is again output. If this occurs the user should again verify that the EPROM is mounted correctly and that the correct personality card and switch settings are being used. After correcting the problem, enter the "Y" response once again.

If the correct EPROM configuration is setup, the programming and verification of the EPROM is performed. When programming begins,

POWER BASIC outputs the following message:

```
PROGRAMMING
```

Programming will continue until the entire BASIC application program in memory has been stored into EPROM. When programming is complete, program verification is begun and the following message is output:

```
VERIFYING
```

If the programming and verification procedures are successful, POWER BASIC will issue the following response:

```
MEM BYTE = XXXX
```

Where "XXXX" is the memory byte which was programmed into the current EPROM and is either the phrase "HIGH" or "LOW", representing bits 0-7 or bits 8-15 of the programmed EPROM, respectively.

Programming of the current EPROM is now complete, and POWER BASIC will issue the following prompt if additional EPROM's are required to store the user's application.

```
MOUNT EPROM
EPROM READY? _
```

The PROGRAM command will perform all the byte selection and memory partitioning for the EPROM's until the entire application has been programmed. The above prompt indicates that POWER BASIC is ready to program the next portion of the user's application. The user should remove the current EPROM and insert the next EPROM for programming. As the EPROM's are programmed and removed they should be clearly labeled with their relative memory address and byte designators. This will avoid confusion when inserting them into the memory map of the Development BASIC system.

When programming of the application is complete, POWER BASIC will issue the following response and return to the keyboard mode to await user command/statement entry.

```
PROGRAMMING COMPLETE
*READY
```

A typical sequence of prompts and user responses are shown below for an user's application requiring two TMS2716 EPROM's. All user entries are underlined.

```
PROGRAM                          (enter command)
RUN? Y                             (auto-run option on) MOUNT EPROM
      (mount 1st EPROM)
EPROM READY? Y                   (enter "Y" when ready)
PROGRAMMING
VERIFYING
MEM BYTE = HIGH                  (bits 0-7 programmed)

MOUNT EPROM                      (mount 2nd EPROM)
EPROM READY? Y                   (enter "Y" when ready)
PROGRAMMING
VERIFYING
MEM BYTE = LOW                   (bits 8-15 programmed)

PROGRAMMING COMPLETE
*READY                           (returns to keyboard mode)
```

If the verification procedure detects an error where the memory contents do not match the EPROM contents, POWER BASIC will issue the response:

```
VERIFY ERROR!
MOUNT EPROM
EPROM READY?
```

When a verification error occurs, POWER BASIC resets all pointers to the beginning of the defective EPROM and awaits the "Y" response to the "EPROM READY?" prompt. The user should remove the TMS2716 EPROM from the programmer and replace it with another erased EPROM. When ready, enter the "Y" response, and the program segment where the error occurred will be programmed again into the new EPROM.

NOTE

> The PROGRAM command is designed to program the entire user application currently in memory from the beginning to the end, performing all byte and EPROM boundary partitioning. Note that an intermediate starting point cannot be specified. For this reason the user typically should not exit the EPROM programmer when a verification error occurs unless the entire EPROM set is to be reprogrammed.

The sequence below illustrates the procedure, prompts, and user responses when programming an application is programmed and a programming error occurs. All user entries are underlined.

```
PROGRAM                          (enter command)
RUN? N                           (audo-run option off)
MOUNT EPROM                      (mount 1st EPROM)
EPROM READY? Y                   (enter "Y" when ready)
```

```
PROGRAMMING
VERIFYING
MEM BYTE = HIGH                 (bits 0-7 programmed)

MOUNT EPROM                     (mount 2nd EPROM)
EPROM READY? Y                  (enter "Y" when ready)
PROGRAMMING
VERIFYING
VERIFY ERROR!                   (verification error)

MOUNT EPROM                     (mount new EPROM)
EPROM READY? Y                  (enter "Y" when ready)
PROGRAMMING
VERIFYING
MEM BYTE = LOW                  (bits 8-15 programmed)

PROGRAMMING COMPLETE
*READY                          (return to keyboard mode)
```

Note that the PROGRAM command may be aborted an any point during programming or verification of the POWER BASIC application program by entering the ESCape (or BREAK) key. When the ESCape key is entered, programming is terminated, the following message is output, and POWER BASIC returns to the keyboard mode to await user command/statement entry.

```
PROGRAMMING TERMINATED
*READY
```

Evaluation BASIC: The PROGRAM command is not supported by Evaluation BASIC.

4.7 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.

4.8 SAVE COMMAND

Forms:

SAVE
SAVE <exp>

The SAVE command writes the source form of the entire POWER BASIC program currently in memory to the cassette device specified by the expression. The program remains in memory after the SAVE and can be deleted by the NEW command. The program may later be retrieved by the LOAD command at some future date.

The SAVE command without an expression, or with an expression value of zero, will result in the user's BASIC program being stored on the 733 ASR digital cassette. Reference section 2, paragraph 2.8.1 for details on 733 ASR digital cassette transport loading and operation. At any time during program saving to digital cassette, the user may terminate the SAVE process by entering the ESCape key.

The SAVE command with an expression value of 1 or 2 will save the program audio cassette drive #1 or #2, respectively. Reference Section 2, paragraph 2.8.2 for audio cassette transport loading operation. Note that the audio cassette device service routines of POWER BASIC reside in the TM 990/452 Development BASIC Enhancement Software Package as presented in Section 1, paragraph 1.1.3 Attempts to execute the "SAVE 1" or"SAVE 2"commands without the Development BASIC Enhancement EPROM set installed in the system will result in an ERROR 41 (ROM installed).

NOTE

> The audio cassette device service routines cannot be interrupted during the saving of a program since each bit of the data bytes have a specified minimum and maximum pulse width for reliable data storage and retrieval. Therefore, all interrupts are masked at the CPU whenever a SAVE is being performed to device 1 or 2. This implies that the real-time clock of POWER BASIC will not be updated for the entire SAVE process. This time period can accumulate to a significant amount. Therefore, the real-time clock is stopped and zeroed when the SAVE process (to audio cassetes) is begun to emphasizes the resulting clock innaccuracy.

Entering a SAVE command with an expression value other than 0, 1 or 2 will result in an ERROR 16 (INVALID DEVICE NUMBER).

Examples:

```
SAVE                (SAVE to 733 ASR digital cassette)
SAVE 0              (SAVE to 733 ASR digital cassette)
SAVE 2              (SAVE to audio cassette drive #2)
```

Evaluation BASIC: The SAVE command with expression parameter is not supported by Evaluation BASIC.

4.9 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. The PRGM size is the current user program size including overhead. This indicates the number of bytes that will be programmed into EPROM if the PROGRAM command of Development BASIC is used. The required overhead of Development BASIC results in a minimum PRGM size of 18 bytes.

Example:

```
SIZE
PRGM:  18 BYTES
VARS:  0 BYTES
FREE:  2012 BYTES
```

SECTION V

BASIC STATEMENTS

5.1 GENERAL

This section discusses 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 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

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
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
READ	Internal Input	Reads from internal data block
DATA	Internal Input	Defines internal data block

TABLE 5-1 (cont)

BASIC PROGRAM STATEMENTS

STATEMENT	FUNCTION	USES
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
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

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. Note that in Evaluation BASIC, the textual information of the REM statement should not exceed 70 characters, or else an error will result. 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 3.4.4.

Examples:

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

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]...)  [,...]>
              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 (4 bytes long in Evaluation BASIC) can contain up to five characters (three characters in Evaluation BASIC). Dimensioned string variables can contain up to the number of elements times 6 minus 1 characters in Development BASIC and 4 times the number of elements minus 1 in Evaluation BASIC. Therefore, the dimensioned string variable $CAT can contain up to 65 characters in Development BASIC and 43 characters in Evaluation BASIC.

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 and 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 6. 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 or string scalar 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.

Examples:

```
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)
```

Evaluation BASIC: The DEF statement is not supported by Evaluation BASIC.

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>
              LET <variable>=<expression>
<line number><variable>=<expression>
             <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.

```
      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"
```

5.6 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 specified line number and continues in sequence with the statements that follow.

Form:

```
<line number> GOTO <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)>
              IF <expression>  THEN<BASIC statement(s)>
<line number> IF <expression><relation><expression> THEN<BASIC statement(s)>
              IF <expression><relation><expression> THEN<BASIC statement(s)>
<line number> IF <string><relation><string> THEN<BASIC statement(s)>
              IF <string><relation><string> THEN<BASIC statement(s)>
<line number> IF <string> THEN<BASIC statement(s)>
              IF <string> THEN<BASIC statement(s)>
<line number> IF <string><relation><string><,expression> THEN<BASIC statement(s)>
              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 BASIC statements depending upon the true or false condition of the last executed IF statement.

Form:

<line number> ELSE <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

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
```

Evaluation BASIC: The ELSE statement is not supported by Evaluation BASIC.

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.

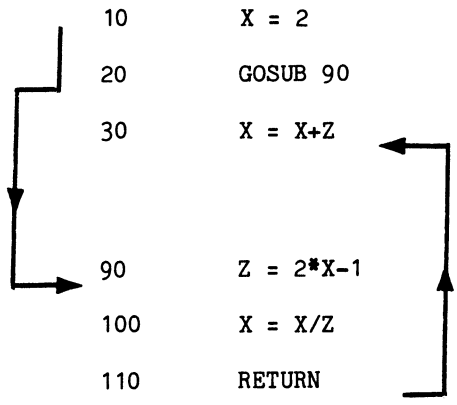


FIGURE 5-1

GOSUB Example

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.

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 Development BASIC (10 levels in Evaluation BASIC). A return address (first line number after the call) 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<>0 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 above 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 in turn calls subroutine 200 until there is no more data. Subroutine 200 exits with a RETURN when data is found and a POP then RETURN when there is no more data. Program execution then continues at line 30.

Evaluation BASIC: The POP statement is not supported by Evaluation BASIC.

5.6.4 ON STATEMENT

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

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

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 the fourth statement number (46) is executed next. 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.

Evaluation BASIC: The ON statement is not supported by Evaluation BASIC.

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 the program loop.

The FOR statement may have either of the following forms:

```
<line number> FOR <variable> = <expression> TO <expression>
              FOR <variable> = <expression> TO <expression>
<line number> FOR <variable> = <expression> TO <expression> STEP <expression>
              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>
              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.

Examples:

```
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:

```
10  REM AREA OF A TRIANGLE
20  FOR B=6 TO 9
30    FOR H=11 TO 13 STEP 0.5
40     A=B*H/2
50     PRINT B,H,A

60    NEXT H
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 Development BASIC and 5 for Evaluation 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 via a GOSUB statement whenever any POWER BASIC error occurs.

Form:

```
<line number> ERROR <line number>
              ERROR <line number>
```

The ERROR statement enables the user to trap to an internal error processing routine on the occurrence of any error. When an ERROR statement has been executed and an error occurs, control passes to the specified line number via a GOSUB statement. The statement number where the error occurred will be placed on top of the GOSUB stack so that if the error is recoverable, a RETURN statement will resume execution at the same statement where the error occurred after it is 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. 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.

Example:

```
10      ERROR 1000
          .
          .
          .

1000    IF SYS(1)=10  THEN PRINT "STROAGE OVERFLOW"::STOP
1010    IF SYS(1)=23 THEN RESTOR::RETURN  (rewind data file)
1020    ELSE PRINT "ERROR=" SYS(1):: STOP
1030    RETURN
```

Statement 100 designates the subroutine starting at statement 1000 to be the error handling subroutine. When an error occurs, control is transferred to statement 1000, and the error number is first tested for "storage overflow". If "storage overflow" is not the error, it is tested for the "read out of data" error number. If this is true, the data is restored to its beginning and control returns to the statement in which the error occurred. If this still was not the error, the error number is output and execution stops.

Evaluation BASIC: The ERROR statement is not suported by Evaluation BASIC.

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

The system displays the line where program execution terminated.

Example:

```
900  STOP
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. The system displays the statement number where program execution terminated.

Example:

```
70  END
STOP AT 70
```

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>

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.

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, 3.14159,"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 (*ERROR 23 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 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 is used to move to a specific point in the data list or to the beginning of the list. 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 were 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 statement, the use of the RESTOR statement does not affect the program.

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 illegal number is entered 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 statement; therefore, the carriage return/line feed is suppressed after entering the constant 12 so that "X SQUARED= 144" can be output on the same line. In line 30 a semicolon is not present so the carriage return/line feed is performed.

When the semicolon is placed before an assignment variable in the INPUT list, the automatic prompting of a question mark or colon is suppressed. The user may then perform his own prompting in the POWER BASIC Program by using PRINT statements or placing character strings 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 line 20 no prompt was issued.

The user may limit the number of characters which can be entered from the keyboard for both numeric and string variable assignments by using the # or % operators in the INPUT statement. Use of the # operator will specify the maximum number of characters which can be entered from the keyboard. Use of the % operator will specify the exact 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 wait to 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  PRINT "YOUR PHONE NUMBER IS";A;"-";B;"-";C
40  STOP

RUN

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

 STOP AT 40
```

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 then prints the user's phone number using the variables of the INPUT list.

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 a GOSUB statement if a control character or invalid input is encountered during input. 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. For example, to the user who requires additional information for the input of data, (control) H can be used to transfer to a routine which outputs a HELP message.

Example:

```
10   INPUT ? 100,N
20   PRINT N
       .
       .
       .
       .
100  REM SUBROUTINE TO PROCESS (control) H INPUT
110  PRINT "USER INPUT ASSISTANCE"
       .
       .
       .
       .

RUN
 ? (control) H
 USER INPUT ASSISTANCE
       .
       .
       .
       .
```

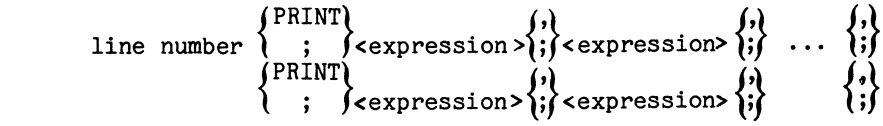
In line 10 if the user enters a numeric value, it will be entered in the variable N; or if the (control) H key is entered, the subroutine at statement 100 will be executed and output the instructions for user input.

Evaluation BASIC: The % and ? operators are not supported by Evaluation BASIC.

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:



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. Evaluation BASIC does not support the direct output of ASCII characters.

Example:

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

would generate

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

Evaluation BASIC does not support direct (<>) entry of control characters; however, any ASCII character can be inserted into string variables with the use of the % operator. The % operator places the single byte value of the succeeding expression into the specified character string. The byte value to be inserted into the string represents the decimal equivalent of an ASCII character. Hexadecimal ASCII character codes may be entered in Development BASIC by using the proper hexadecimal notation (e.g., $A=%0AH%0DH%0H). Byte value insertion should always be terminated with a null (zero byte) insertion (e.g., %0AH%041H%0H). ASCII codes may be concatented to character strings, however charcter 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  $A(0)="GOTO THE NEXT LINE"+$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 should be of the type which buffers the characters and automatically generates a carriage return/line feed when its buffer is full to obtain the correct five column output. If the terminal device does not perform in this manner, output values may be lost at the end of output lines, and the five column output format may be skewed. 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 most 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=  10       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!
```

Evaluation BASIC: The ability to directly output hexadecimal ASCII codes (e.g.,<0A>) is not implemented in Evaluation BASIC. However, the % operator may be used to perform this function as explained earlier in this section. It will insert the byte code corresponding to the decimal value of the expression following the % operator.

Example:

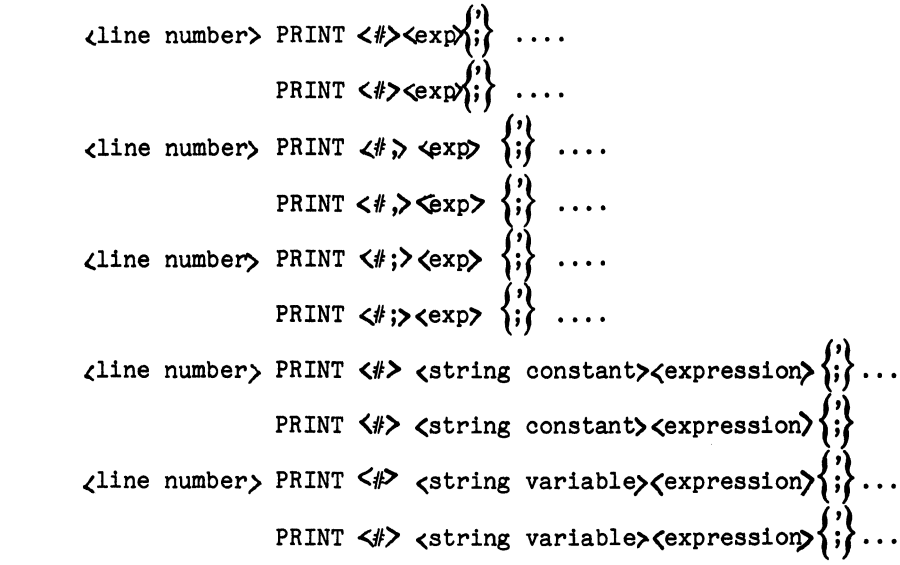
```
10  $A=%10%0   !LINE FEED (0A)=10
20  PRINT $A
```

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. Print formatting is not supported by Evaluation BASIC.

Print formatting using the formatting string for decimal numeric output is supported by Development BASIC only when the TM 990/452 ENHANCEMENT SOFTWARE PACKAGE EPROM set is installed in the system as presented in Section 2, Paragraph 2.3.2. Execution of PRINT statement utilizing this option without the Development Support option EPROM set being installed will result in an "EXPANSION EPROM NOT INSTALLED" error.

Forms:



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 hte 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

T A B L E 5 - 2

FORMATTING STRING CHARACTERS

CHARACTER	FUNCTION	EXAMPLE
.	Decimal point specifier	PRINT #"999.99"25.32; ⌀25.32
^	Translates to decimal point	PRINT #"999^00"1000; ⌀10.00
,	Suppressed if before significant digit	PRINT #"99,999.99"100; ⌀⌀⌀100.00
9	Digit holder	PRINT #"9999"123; ⌀123
0	Digit holder or forces zero	PRINT #"9999.999".234; ⌀⌀⌀0.234
$	Digit holder & floats $	PRINT #"$$$.99"8; ⌀$8.00
S	Digit holder & floats sign	PRINT #"SSS.99" -6; ⌀-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

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. In practice these quotes typically would not be used. The user who has Development BASIC may execute these examples from the keyboard by entering the example through the final semicolon (;), inclusive, and then terminating the entry line with a carriage return. 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 "0" 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>'
```

Evaluation BASIC: Print formatting is not supported on Evaluation BASIC.

Development BASIC: Note that decimal print formatting utilizing a formatting string is supported by Development BASIC only when the TM 990/452 Enhancement Software Package EPROM set is installed in the system.

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 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.

```
<exp>  }      May not appear together without a separator
<var>  }      between them.
<$var> }
" string "  }
TAB         }
 ,          }   Separators
 ;          }
 #          }
```

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 UNIT STATEMENT

The UNIT statement designates the device or devices to which all subsequent printed output will be sent.

Forms:

```
<line number> UNIT <expression>
              UNIT <expression>
```

The expression may be any numeric constant, variable, or expression which is evaluated and its integer portion is used. The UNIT statement is used in conjunction with Development BASIC using a TM 990/101M microcomputer board. The UNIT statement directs all printed output to either or both of the serial interfaces present on the TM 990/101M. Note that the TM 990/101M microcomputer board has only one serial I/O port, and therefore the UNIT statement of Development BASIC will only affect the output to port A of this board.

The unit number assignments are as follows:

UNIT	SERIAL I/O PORT
1	SERIAL PORT A (CRU = 0080_{16})
2	SERIAL PORT B (CRU = 0180_{16})
3	SERIAL PORTS A & B

All output will be directed to the device on the selected I/O port, including LIST output, BASIC command/statement output, and all program generated output. The serial ports will remain selected for output until a subsequent UNIT statement is encountered.

Examples:

```
100     UNIT   1              (directs output to PORT A)
200     UNIT   3              (directs output to PORTS A & B)
```

Evaluation BASIC: The UNIT statement is not supported by Evaluation BASIC.

5.8.4 BAUD STATEMENT

The BAUD statement is used to set the baud rate of the serial I/O port(s) in either the program on keyboard mode.

Forms:

```
<line number><BAUD <expression 1>,<expression 2>
             <BAUD <expression 1>,<expression 2>
```

The BAUD statement will initialize the TMS9902 Asynchronous Communications 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 TM 990/100M or TM 990/101M microcomputer board, while a non zero value will select port B (CRU address> of 180) of the TM 990/101M 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:

```
     BAUD   0,4      (in Keyboard mode)
10   BAUD   0,0
20   BAUD   1,2
```

Evaluation BASIC: The BAUD statement is not supported by Evaluation BASIC.

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

Evaluation BASIC: Interrupt processing is not supported by Evaluation BASIC.

5.9.1 IMASK STATEMENT

The IMASK statement is used to control the interrupt mask of the TMS9900 microprocessor. The TMS9900 microprocessor employs 16 interrupt levels with the highest priority level 0 and the lowest level 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.

Examples:

```
10   IMASK  15   !  SET MASK TO 15
20   IMASK  OEH  !  SET MASK TO 14
30   A=OAH       !  SET A TO 10
40   IMASKA      !  SET MASK TO VALUE OF A
```

Evaluation BASIC: The IMASK statement is not supported by Evaluation BASIC.

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" 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 cannot 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
```

Evaluation BASIC: The TRAP statement is not supported by Evaluation BASIC.

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.

Example:

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

Evaluation BASIC: The IRTN statement is not supported by Evaluation BASIC.

5.9.4 Assembly languge processors

There are times when it may be necessary or advisable for the interrupt processor to be written in assembly language. This may be

accomplished in two ways in POWER BASIC. 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, address as 0 through 3F, is reserved for 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 a subroutine to serve the interrupt.

The reserved memory locations are shown in 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.

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.

To install an assembly language interrupt processor, the user must create a data set containing the contents of the low memory EPROM set (U42 and U44). The transfer vector for the desired interrupt level must be modified to reflect the new workspace pointer and the new entry point for the interrupt routine. A new EPROM set must then be programmed from this data set. This new EPROM set will replace the original set and should be mounted in the EPROM sockets on the processor board (U42 and U44). The EPROM set containng the user's interrupt handler must then be mounted at the desired address.

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 available contiguous RAM for its own use. Consequently, the user must either supply non-contiguous RAM for the workspaces or must use the memory option of the NEW command to deallocate the required RAM after auto sizing.

If the workspace area is allocated by the NEW command, it must be done each time that POWER BASIC is restarted and prior to the entry of a Basic Application program.

Note that an 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. That is, 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.

5.10 BASE STATEMENT

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

Form:

```
<line number> BASE <expression>
              BASE <expression>
```

The BASE statement evaluates the expression and sets the CRU base address to the result for use by the CRB and CRF functions. The CRB function addresses bits within +127-128 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 set, display, or store the 24 hour time of day clock.

Forms:

```
<line number> TIME  <exp>,<exp>,<exp>
              TIME  <exp>,<exp>,<exp>
<line number> TIME <string variable>
              TIME <string variable>
<line number> TIME
              TIME
```

The TIME statement is used with the expression list to set and start the time of day clock. The form of the expression is as follows:

```
TIME HH,MM,SS
```

where

H = hours, M = minutes, S = seconds

The clock is set up as a 24-hour clock with times ranging from 00:00:00 to 23:59:59. Initialization of the clock is valid at any point in the program. Its value may also be reinitialized at any point.

Examples:

```
        TIME 10,27,30  (in keyboard mode)
        TIME 3,5,0  (in keyboard mode)
10      TIME  21,8,15
```

The second form of the TIME statement enables storing the current time of day into 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 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 third form of the TIME statement. The time of day will be displayed in the following format:

```
HH:MM:SS
```

Note that when the user executes a SAVE or LOAD on audio cassette in Development BASIC (SAVE 1 or 2, or LOAD 1 or 2), all interrupts are masked at the CPU because the audio cassette software cannot be interrupted since each byte has specified minimum and maximum bit times for reliable data storage and retrieval.

This implies that the real-time clock will not be updated for the entire LOAD or SAVE process. This time period can accumulate to a significant amount, therefore the real-time clock is stopped and cleared when using audio cassettes to emphasizes the resulting clock innaccuracy.

Examples:

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

RUN
 11:04:37

 STOP AT 110
```

5.12 RANDOM STATEMENT

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

Forms:

```
<line number>  RANDOM  <expression>
               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 RND
30  RANDOM RND * MEM(X)
```

Evaluation BASIC: The RANDOM statement is not supported by Evaluation BASIC.

5.13 ESCAPE AND NOESCAPE STATEMENTS

The ESCAPE and NOESC statements provide capability to enable or disable the escape key to interrupt program execution.

Forms:

```
<line number> ESCAPE
<line number> NOESC
```

The ESCAPE statement enables the terminal device escape (or break) key to interrupt program execution. When the escape key is struck program execution 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 switch on the CPU board.

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

The ESCAPE statement is used during program development and debug. The NOESC statement is used for time critical application programs or in a production environment where it is disadvantageous for the user to interact with POWER BASIC in a non-program controlled mode.

Examples:

```
10 ESCAPE
10 NOESC
```

Evaluation BASIC: The ESCAPE and NOESC statements are not supported by Evaluation BASIC. The terminal device escape key is always enabled in Evaluation BASIC.

5.14 CALL STATEMENT

The CALL statement allows the user access to assembly language subroutines. The user may pass up a 4 parameters to the subroutine although only 1 is required.

Forms:

<line number> CALL <string-constant>,<address> [,<var 1>,<var>2,<var>3, <var>4]

CALL <string-constant>,<address> [,<var>1 ,<var>2,<var>3, <var>4]

where:

string constant is the IDT name of the assembly language subroutine

address is the hexadecimal address at the assembly language subroutine

var 1, var 2, var 3, and var 4, are the parameters of the subroutine

String constant is the IDT name of the assembly language subroutine being called. The hexadecimal address is the location in memory at which the assembly language routine resides. A string constant must be entered to execute a CALL statement, although only the "address" is used by Development BASIC. A string constant must be entered to execute a CALL statement, although only the "address" is used by Development BASIC. A string constant maintains transportability of the CALL statement to configurable BASIC.

If the parameter is passed as a value, it will be converted into a 16-bit two's complement integer. If passed by address, no conversion takes place, and the value must be interpreted as a two word integer or three word floating point value beginning at the address pased in the register. Reference Section 3.7.6 for detailed information on internal variable formats.

Assembly language routines may only use Registers 4,5,6, and 7 of the POWER BASIC workspace. Therefore all assembly language routines must supply their own workspaces (typically by executing a BLWP as the first statement of the assembly language routine), and require a RAM area to be allocated for this purpose. During Power-up RESET, POWER BASIC will automatically size available contiguous RAM for its own use from the top of memory (FFFE) down until a write/read sequence results in a mismatch. The user must either supply non-contiguous RAM for the workspace, or use the memory option of the NEW command to deallocate the required RAM after auto-sizing. Note that if the workspace area is allocated by the NEW command, the NEW command mut be executed each time that POWER BASIC is powered up on re-initialized (via the RESET switch) and prior to the entry of a BASIC application program.

For example:

```
CALL "SAMPLE", 0E000H, A, (B)
```

will result in a branch and link to the subroutine "SAMPLE" location E000 (hex) with the value of A (converted to 16 bit two's complement integer) passed in R4 and the address of the variable B is passed in R5.

Evaluation BASIC: The CALL statement is not supported by Evaluation BASIC.

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:

Development BASIC
    Number of characters = 6 x (number of variable elements) - 1

Evaluation BASIC
    Number of characters = 4 x (number of variable elements) - 1

Examples:

I1	6 x 1 - 1 = 5	(Development BASIC)
I1	4 x 1 - 1 = 3	(Evaluation BASIC)
A(10)	6 x 11 - 1 = 65	(Development BASIC)
A(10)	4 x 11 - 1 = 43	(Evaluation BASIC)
N(10,5)	6 x (11 x 6) - 1 = 395	(Development BASIC)
N(10,5)	4 x (11 x 6) - 1 = 263	(Evaluation BASIC)

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 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 dollar sign preceding a variable or a literal string. $<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 corresponding DATA statement entry is also an ASCII variable or an ASCII string. When data types do not match you receive an error at the line number 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 DATA", the variable A receives the number 12345 and B the number 123450. Finally, the ASCII variable $Z(0) receives the same string as $N(0).

A dimensioned variable can have a byte index by following the subscripts with a semicolon and the byte displacement. The range of the index is from 1 through the last byte 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),C(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 together 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),B(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.

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
40  $B(0;5)=$A(0);1
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),B(10)
20  $A(0)="ABCDEFG"
30  $A(0;4)=/"..."
40  PRINT $A(0)
50  STOP

RUN
 ABC...DEFG

 STOP AT 50
```

Evaluation BASIC: Note that character insertion is not supported by Evaluation BASIC.

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),B(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  $A (0;5)= /10
40  PRINT $A(0)
50  STOP

RUN
 ABCDOPQRSTUVWXYZ

 STOP AT 50
```

Evaluation BASIC: Character deletion is not supported by Evaluation BASIC.

6.8 BYTE REPLACEMENT

Individual bytes may be altered by using the decimal equivalent of an ASCII character along with the "%" operator.

Form:

```
$<VAR> = %<EXP> ...
```

The evaluated expression specifies the byte code to replace in the string variable. Byte replacements may be chained together.

Example:

```
10  DIM A(10),B(10)
20  $A(0)=%65%66%0
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:

```
<VAR> = <$VAR> , <VAR>
```

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  N="1234",E
20  N1="12DE",E1
30  PRINT N,$E
40  PRINT N1,$E1
50  STOP

RUN
 1234
 12                D
STOP AT 50
```

Evaluation BASIC: ASCII to number conversion is not supported by Evaluation BASIC.

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 same formatting rules are followed as 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
```

Evaluation BASIC: Numeric to character conversion is not supported by Evaluation BASIC.

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
```

Evaluation BASIC: The string variable length is not supported by Evaluation BASIC.

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 of where the first string is located in the second string. If the search was unsuccessful, a zero is returned.

Example:

```
10  DIM A(0),B(0)
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
```

Evaluation BASIC: The search function is not supported by Evaluation BASIC.

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),B(10)
20  $A(0)="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30  PRINT MCH ("ABCDXYZ",$A(0)),MCH("BC",$A(0;2))
40  STOP
```

```
RUN
 4          2

STOP AT 40
```

Evaluation BASIC: The character match function is not supported by Evaluation BASIC.

.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.

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   D=ASC($C)
60   IF D=020H THEN $B(0;I)=%D%0
70     ELSE $B(0;I)=%(D+020H)%0
80  NEXT I
90  PRINT $A(0)
100  PRINT $B(0)
110  GOTO 20

RUN
INPUT STRING:  UPPER CASE TO LOWER CASE
UPPER CASE TO LOWER CASE
upper case to lower case
INPUT STRING
```

Evaluation BASIC: The ASCII character conversion function is not supported by Evaluation BASIC.

SECTION VII

POWER BASIC FUNCTIONS

7.1 GENERAL

POWER BASIC includes several predefined mathematical, string, and miscellaneous functions. 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 7.2.7) 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 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
```

Evaluation BASIC: The ABS function is not supported by Evaluation BASIC.

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
 80.41951
```

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
 0.9489849   0.3153215
```

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
 7.20049E+10
```

Evaluation BASIC: The EXP function is not supported by Evaluation BASIC.

7.2.5 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.

Example:

```
10  INPUT Y
20  IF INP(Y/2)<>Y/2 THEN GOTO 50
30  PRINT "Y IS AN EVEN NUMBER"
40  STOP
50  PRINT "Y IS AN ODD NUMBER"
60  STOP
```

Executing the previous example produces:

```
? 75
Y IS AN ODD NUMBER
```

7.2.6 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.

Example:

```
10  INPUT L
20  PRINT LOG(L)
30  STOP
```

Executing the above example produces:

```
? 5280
 8.57168
```

Evaluation BASIC: The LOG function is not supported by Evaluation BASIC.

7.2.7 SQUARE ROOT FUNCTION (SQR)

The square root (SQR) function 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 (*ERROR 25 AT XXXX) is produced if the argument is negative.

Example:

```
10  INPUT K
20  PRINT SQR(K)
30  STOP
```

Executing the above example produces:

```
? 2
 1.414214
```

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 (ASC) function returns the decimal ASCII numeric value of the first character of the specified string function. 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
```

Evaluation BASIC: The ASC function is not supported by Evaluation BASIC.

7.3.2 LENGTH FUNCTION (LEN)

The length (LEN) function returns the number of non-null characters starting at the evaluated 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
```

Evaluation BASIC: The LEN function is not supported by Evaluation BASIC.

7.3.3 CHARACTER MATCH FUNCTION (MCH)

The match (MCH) function 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)
```

Evaluation BASIC: The MCH function is not supported by Evaluation BASIC.

7.3.4 CHARACTER SEARCH FUNCTION (SRH)

The search (SRH) function 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     (RESULTS)
```

Evaluation BASIC: The SRH function is not supported by Evaluation BASIC.

7.4 MISCELLANEOUS FUNCTIONS

The miscellaneous functions described in paragraphs 7.4.1 through 7.4.4 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 are will be transferred. For example:

```
CRF(0) = -1
```

transfers 16 bits (hex 'FFFF') 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 (NKY) function 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(65) THEN PRINT "A"
```

prints "A" if the last key entered was "A". The argument value is expressed in decimal.

7.4.4 SYSTEM INTERROGATION (SYS) FUNTION

The system interrogation (SYS) function 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.)

```
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.

Evaluation BASIC: The SYS function is not supported by Evaluation BASIC.

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 utilizes the real time clock of the TMS9901, programmed to generate an interrupt (or TIC) every 40 milliseconds (1/25th of a second) when the system clock rate is at 3MHz.

The following example will output the ASCII code "07" corresponding to the bell on the terminal once each second.

```
LIST
    10  TIME 0 ! THIS WILL START THE CLOCK
    20  $B=%7%0!  THIS IS THE CODE FOR CONTROL 'G' (BELL)
    30  A=TIC(0)
    40  IF TIC(A) <> 25 THEN GOTO 40
    50  PRINT $B;
    60  GOTO 30
```

7.4.6 MEMORY MODIFICATION (MEM) FUNCTION

The memory modification (MEM) function reads or modifies a memory location (byte) as specified by the argument. 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".

Hexadecimal constants are not supported by Evaluation BASIC; therefore, all arguments must be in decimal form. When addressing memory in decimal notation the memory will be addressed as follows:

Decimal address	Hexadecimal address
0	0000
1	0001
.	.
.	.
.	.
.	.
32767	7FFF
32768	8000
.	.
.	.
.	.
.	.
65534	FFFE
65535	FFFF

Example:

```
M = MEM (65280)
```

reads the byte from location hex FF00, while

```
MEM (65520) = 25
```

stores a decimal 25 at location hex FFF0.

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.

Evaluation BASIC: The BIT function is not supported by Evaluation BASIC.

7.4.8 RANDOM NUMBER (RND) FUNCTION

The RANDOM NUMBER (RND) function 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.

APPENDIX A

ERROR CODES

The following error codes may be issued by the POWER BASIC Executive:

CODE	ERROR MESSAGE
1	SYNTAX ERROR
2	UNMATCHED PARENTHESIS
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 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 = EXPANSION EPROM NOT INSTALLED
42 = INTERRUPT W/O TRAP
43 = INVALID BAUD RATE
44 = TAPE READ ERROR
45 = EPROM VERIFY ERROR
46 = INVALID DEVICE NUMBER

APPENDIX B

STATEMENT AND COMMAND SUMMARY

This Appendix contains a summary of all POWER BASIC statements and commands for the Development and Evaluation BASIC Software Packages. An explanation preceded by an asterisk (*) indicates the statements and commands which are not supported by Evaluation BASIC.

EDIT MODE COMMANDS

An advanced editor is contained in POWER BASIC to aid in program writing, editing, and debugging. The editor uses the following special control characters. Note that the phrase "(ctrl)" indicates tht the user holds down the control key while depressing the key corresponding to the character immediately following.

SYNTAX	EXAMPLE/EXPLANATION
(CR)	(CR) Enter last line typed into program source
(ctrl)In	(ctrl)I4 *Insert n blanks.
(ctrl)Dn	(ctrl)D4 *Delete n characters.
(ctrl)H	(ctrl)H Backspace 1 character.
(ctrl)F	(ctrl)F Foward space 1 character.
ln(ctrl)E	100(ctrl)E Display source line indicated by line number (ln) for editing.
(ctrl)T	(ctrl)T Toggle from one partition to the other partition. (Only in Evaluation BASIC).

SYNTAX	EXAMPLE/EXAMPLANATION
(esc)	Cancel input line or break program execution.
(Rubout) or (DEL)	Backspace and delete character.

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. POWER BASIC commands may be abbreviated to the first three letters of the command name, and all letters must be entered in upper case.

SYNTAX	EXAMPLE/EXPLANATION
CONtinue	CONTINUE
	*Execution continues from last break.
‹ln› LIST	LIST
	List the user's POWER BASIC program. ln LIST - Will list from specified line number through end of program or until the ESCape key is entered.
LOAD	LOAD
	Reads a previously recorded POWER BASIC program from 733 ASR digital cassette.
LOAD exp	LOAD 2
	*Reads a previously recorded POWER BASIC program from audio cassette drive no. 1 or no. 2.
LOAD ‹address›	LOAD 05000H
	*Configures POWER BASIC to execute BASIC programs stored in EPROM at the specified address.
NEW	NEW
	Clear current user program, variables, pointers, and stacks, and prepares for entry of new program.

SYNTAX	EXAMPLE/EXPLANATION
NEW <address>	NEW 0A000H *Sets the lower RAM memory bound used by POWER BASIC after auto-sizing at power-up.
PROgram	PROGRAM *Program current POWER BASIC application program into EPROM.
RUN	RUN Begin program execution at the lowest line number.
SAVE	SAVE Records a POWER BASIC program onto 733 ASR digital cassette.
SAVE <exp>	SAVE 1 *Records a POWER BASIC program or audio cassette drive no. 1 or no. 2.
SIZE	SIZE Display current program size, allocated variable space, and available memory in bytes.

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.

SYNTAX — EXAMPLE/EXPLANATION

ln BAUD <exp1> <,exp2>
BAUD 0,5

*Sets the baud rate of the serial I/O port(s).

ln BASE <(exp)> BASE (256)

Sets CRU base for subsequent CRU operations.

ln CALL <string-constant>, <subroutine address> [,var1] [,var2] [,var 3] [,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.

ln DATA { <exp> / <string-constant> } [, { <exp> / <string-constant> }] ....

DATA 1, 4*ANT(1), "HI"

Define internal data block for access by READ state ment.

ln DEF FN<x> [(<arg1> [,arg2] [,arg3])] =<exp>

DEF FNA (X,Y)=(3*X+Y)/Y

*Defines user arithemetic function

SYNTAX	EXAMPLE/EXPLANATION
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 ERROR ln	ERROR 1000 *Specifies a subroutine that will be called via a 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.

SYNTAX	EXAMPLE/EXPLANATION
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)
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 TMS9900 microprocessor to specified level.

SYNTAX	EXAMPLE/EXPLANATION
ln TRAP <level> TO <ln>	TRAP 9 TO 1000 *(see TRAP statement)
ln IRTN	IRTN *(see IRTN statement)
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 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.

SYNTAX	EXAMPLE/EXPLANATION
ln POP	POP *Removes from the GOSUB stack the last pushed return address without an execution transfer.
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 RETURN	RETURN Return from BASIC subroutine and remove top address from GOSUB stack.

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 expressions 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 UNIT <exp>

UNIT 3

*Designate the device(s) to receive all printed output.

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.

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 the zero if the assigned value (exp2) is zero.
COS (<exp>)	A=COS(B) Cosine of expression. (expression in radians)

SYNTAX	EXAMPLE/EXPLANATION
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)=OFH 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.
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.

SYNTAX	EXAMPLE/EXPLANATION
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. Only decimal values may be used by Evaluation BASIC (valid integer range of 0 to 65535).
MEM (<exp1>)=	<exp2> MEM(OBOH)=OFH MEM(176)=15 Store byte exp2 into user memory at address specified by exp1. Only decimal values may be used by Evaluation BASIC (valid range of 0 to 65535).
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.
RND	A=RND Returns a random number between 0 and 1.
SIN (<exp>)	A=SIN(B) Sine of the expression (exp in radians).

SYNTAX	EXAMPLE/EXPLANATION
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
TIC (<exp>)	T1=TIC (0) T2=TIC(T1) Returns the number of time TICs less the expression value. One TIC equals 40 milliseconds (1/25 second).

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 Development BASIC which are 6 bytes in length, can contain up to 5 characters (Evaluation BASIC has 4 byte variables, so only 3 characters can be entered per simple string variables). Dimensioned string variables in Development BASIC may contain up to the number of elements times 6, less one character, while Evaluation BASIC dimension variables may contain up to the number of elements times 4, 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-var> / <string-constant>}	$A(0)=$B(0) $A(0)=$B(0;5) Store string into string-var 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-var> / <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-var> / <string-constant>}	$A(0;3)=/"HI" Insert string into specified posi tion in string variable.

FUNCTION/SYNTAX	EXAMPLE/EXPLANATION
String Length Function <num-var>=LEN(<string-var> <string-constant>)	L=LEN($A(0)) L=LEN "ABC" Return the length of string up to terminating null.
Pick <string-var1> = {<string-var2> <string-constant2>}, <exp>	$I=$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-var2> <string-constant2>} ; <exp>	$B(0;2)=$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 $A Prompt with colon and input character data.
,	INPUT A,B Delimit expressions for multiple inputs
;	INPUT ;A INPUT A; Suppress prompting if before variable or CR LF if at end of line.
#exp	INPUT #1"Y or N"$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.

"STRING"

```
INPUT "YES or NO?";$A
```

Prompt with string and then get input. Equivalent to:

```
PRINT "YES or NO?";:INPUT;$A
```

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 "<OD><OA>"
	*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.

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

ERROR CODES

POWER BASIC displays error code numbers corresponding to the appropriate error messages listed below. This is the case for the TM 990/450 Evaluation BASIC package and for the TM 990/451 Development BASIC package. However, Development BASIC utilizing the TM 990/452 Development BASIC Enhancement Software Package EPROM set will display the error message itself in place of the error code number for all errors generated by the Development BASIC package.

CODE ERROR MESSAGE

1 = SYNTAX ERROR
2 = UNMATCHED PARENTHESIS
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

CODE ERROR MESSAGE (cont.)

38 = UNDEFINED FUNCTION
39 = UNDIMENSIONED VARIABLE
40 = UNDEFINED VARIABLE
41 = EXPANSION EPROM NOT INSTALLED
42 = INTERRUPT W/O TRAP
43 = INVALID BAUD RATE
44 = TAPE READ ERROR
45 = EPROM VERIFY ERROR
46 = INVALID DEVICE NUMBER

APPENDIX C

SAMPLE PROGRAMS

This Appendix consists of two sample programs (sine wave and word puzzle).

C.1 SINE WAVE

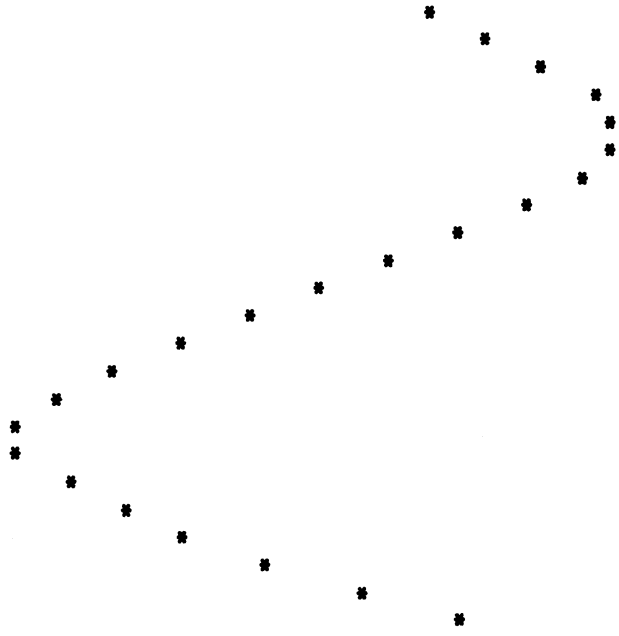
This sample program demonstrates the use of the TAB function. It will plot a sine wave given input from the user.

```
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
```

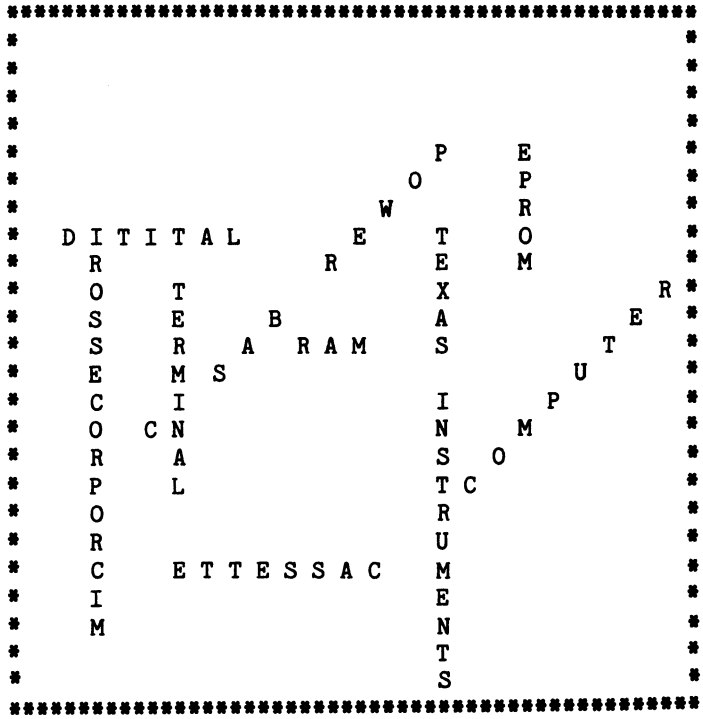
2 WORD PUZZLE

This sample program demonstrates character array manipulation. It will hide up a twenty one user-selected words in an array of random letters. Note that an expansion memory board is required execute this program on both Development and Evaluation BASIC.

```
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 AGAIN
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

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.

D.1 INTRODUCTION

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] b..XOP b.. GA,OP b.. 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  @FP21   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.

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
          .
          .
    FL100 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
```

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: _____ _____
_____ _____
_____
_____
_____
_____
_____
_____
_____
_____

EQUIPMENT 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 DISCREPANCY. USE ADDITIONAL PAGES IF NECESSARY.

MAIL TO:

TEXAS INSTRUMENTS INCORPORATED
P. O. BOX 1443, M/S 6407
HOUSTON, TEXAS 77001

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001
Semiconductor Group

MP308 Rev. *A

Printed in U.S.A.