

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES AND DISCARD SUPERSEDED PAGES

Note: The changes in the text are indicated by a change number at the bottom of the page and a vertical bar in the outer margin of the changed page. A change number at the bottom of the page but no change bar indicates either a deletion or a page layout change.

TI BASIC Reference Manual (2308769-9701)

Original Issue1 February 1983
 Change 11 December 1983

Total number of pages in this publication is 288 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	1	5-1 - 5-22	0	9-7	1
Effective Pages	1	6-1 - 6-6	0	9-8	0
iii	0	6-7 - 6-8	1	10-1	1
iv - vi	1	6-9 - 6-22	0	10-2 - 10-4	0
vii - xiii	0	6-23	1	11-1	1
xiv	1	6-24 - 6-25	0	11-2 - 11-6	0
xv/xvi	0	6-26	1	11-7	1
1-1 - 1-4	0	6-27 - 6-29	0	11-8 - 11-20	0
2-1 - 2-3	1	6-30	1	12-1 - 12-9	0
2-4 - 2-5	0	6-31 - 6-34	0	12-10	1
2-6 - 2-8	1	6-35	1	12-11 - 12-12	0
3-1 - 3-2	0	6-36 - 6-46	0	12-13	1
3-3 - 3-6	1	7-1 - 7-11	0	12-14	0
3-7	0	7-12	1	A-1 - A-14	1
3-8	1	8-1 - 8-18	0	B-1 - B-6	0
3-9 - 3-14	0	8-19	1	C-1 - C-2	0
4-1	1	8-20	0	D-1 - D-5	0
4-2 - 4-4	0	8-21	1	D-6	1
4-5 - 4-6	1	8-22	0	D-7 - D-18	0
4-7 - 4-8	0	8-23	1	E-1	1
4-9	1	8-24 - 8-26	0	E-2 - E-4	0
4-10	0	9-1 - 9-6	0	F-1 - F-4	0

©Texas Instruments Incorporated 1983
 All Rights Reserved, Printed in U.S.A

The information and/or drawings set forth in this document and all rights in and to inventions disclosed herein and patents which might be granted thereon disclosing or employing the materials, methods, techniques or apparatus described herein, are the exclusive property of Texas Instruments Incorporated.

Preface

This manual describes the Texas Instruments BASIC* language, hereafter referred to as BASIC. BASIC is a symbolic programming language oriented toward financial, engineering, and scientific applications. This manual explains in detail the correct syntax of program statements and operating commands. Although BASIC is an excellent tool for learning to write computer programs, you are assumed to have some familiarity with computer programming and its concepts before beginning this manual. For tutorial reference, TI offers a BASIC self-study manual, *TI BASIC, A Tutorial Approach to Programming*, part number 2305790-9701. Many other commercial textbooks and guides are also available.

Before using BASIC, you must have installed TI BASIC on your operating system following the directions in the BASIC Installation guide. This manual assumes that you know how to turn on and turn off your system and are familiar with the System Command Interpreter (SCI) file creation commands described in your operating system programmer's guide.

This manual is organized into the following sections and appendixes.

Section

- 1 General Description — Describes the components of TI BASIC and the TI enhancements to ANSI-standard BASIC.
- 2 Getting Started — Describes the process of writing, saving, and executing a BASIC program and provides a simple example.
- 3 BASIC Commands — Presents the commands used to manage the system resources.
- 4 Editing Capabilities — Describes the BASIC editor; emphasizes the use of keyboard editing functions in program development and describes the use of the OPTION statement to inhibit the function keys during program execution.
- 5 Data and Expressions — Describes the types and attributes of data on which BASIC operates and how the data types can be combined to form expressions.
- 6 Input/Output (I/O) Statements — Describes in detail the I/O facilities of BASIC, including formatting.
- 7 Control Statements — Describes conditional and unconditional transfer of control within a program.
- 8 Intrinsic Functions — Describes the built-in arithmetic, string, time and date, and miscellaneous functions of BASIC.

* BASIC is a trademark registered by the trustees of Dartmouth College, Hanover, New Hampshire.

- 9 User-Defined Procedures — Explains how to define functions and subprograms.
- 10 Debug Features — Describes commands and methods useful in locating program errors.
- 11 Assembly Language Subroutines — Describes the process of linking BASIC to assembly language subroutines.
- 12 BASIC Subroutine Library — Describes the sort and key file package (KFP) subroutines.

Appendix

- A Keycap Cross-Reference — This appendix contains specific keyboard information to help the user identify individual keys on any supported terminal.
- B ASCII Character Set — Lists the ASCII and graphics characters supported by TI BASIC.
- C BASIC Reserved Word List — Lists the words reserved for the TI BASIC language.
- D Error Messages and Codes — Lists the BASIC error messages and codes.
- E BASIC System Differences — Describes the system differences between DX10 Micro, DX10, and DNOS BASIC.
- F Logical Operators with Integer Operands — Describes the use of logical operators within the BASIC systems.
- G Syntax Diagrams — Describes the syntax of the individual BASIC statements in a schematic format.
- H Example Programs — Provides examples of programs illustrating important features of BASIC.
- I BASIC Systems Function Keys — Lists the functions supported by BASIC and specifies the keys that control these functions on various TI keyboards.

Consult the appropriate manual for more information about your operating system and other system software:

Title	Part Number
DX10 Manuals	
<i>DX10 Operating System Computer Concepts and Facilities (Volume I)</i>	946250-9701
<i>DX10 Operating System Operations Guide (Volume II)</i>	946250-9702
<i>DX10 Operating System Application Programming Guide (Volume III)</i>	946250-9703

Title	Part Number
<i>DX10 Operating System Text Editor (Volume IV)</i>	946250-9704
<i>DX10 Operating System Systems Programming Guide (Volume V)</i>	946250-9705
<i>DX10 Operating System Error Reporting and Recovery Manual (Volume VI)</i>	946250-9706
DNOS Manuals	
<i>DNOS Concepts and Facilities Manual</i>	2270501-9701
<i>DNOS Operations Guide</i>	2270502-9701
<i>DNOS System Command Interpreter (SCI) Reference Manual</i>	2270503-9701
<i>DNOS Text Editor Reference Manual</i>	2270504-9701
<i>DNOS Messages and Codes Reference Manual</i>	2270506-9701
<i>DNOS Systems Programmer's Guide</i>	2270510-9701
<i>DNOS System Generation Reference Manual</i>	2270511-9701
<i>DNOS Link Editor Reference Manual</i>	2270522-9701
Business System 200 Manuals	
<i>Business System 200 Operator's Guide</i>	2533266-9701
<i>DX10 Micro Messages and Codes</i>	2533322-9701
<i>DX10 Micro User's Guide</i>	2228263-9701
<i>DX10 Micro Operating System Handbook</i>	2533262-9701
Other Manuals	
<i>990/99000 Assembly Language Reference Manual</i>	2270509-9701
<i>TMS 9900 Microprocessor Assembly Language Programmer's Guide</i>	943441-9701
<i>Link Editor Reference Manual</i>	949617-9701

Contents

Paragraph	Title	Page
1 — General Description		
1.1	Introduction	1-1
1.2	BASIC System Configuration	1-2
1.3	Manual Overview	1-3
1.4	Program Development	1-4
1.4.1	Writing the BASIC Program	1-4
1.4.2	Listing the Program	1-4
1.4.3	Executing the Program	1-4
1.4.4	Editing the Program	1-4
1.4.5	Correcting the Program	1-4
1.4.6	Saving the Program	1-4
2 — Getting Started		
2.1	Initial Operations	2-1
2.2	Sample Program	2-2
2.2.1	Step 1 — Enter BASIC	2-3
2.2.2	Step 2 — Create the Program	2-3
2.2.3	Step 3 — Analyze the Coding	2-6
2.2.4	Step 4 — Execute the Program	2-7
2.2.5	Step 5 — Terminate BASIC	2-7
3 — BASIC Commands		
3.1	Introduction	3-1
3.2	NEW Command	3-2
3.3	NUMBER Command	3-2
3.4	RESEQUENCE Command	3-3
3.5	OLD Command	3-5
3.6	EDIT Command	3-5
3.6.1	EDIT MAIN	3-5
3.6.2	EDIT ESUB	3-5
3.7	LIST Command	3-6
3.8	DELETE Command	3-6
3.8.1	DELETE File	3-7
3.8.2	DELETE ESUB	3-7
3.8.3	DELETE Lines	3-8
3.9	RENAME Command	3-9

Paragraph	Title	Page
3.10	MERGE Command	3-9
3.11	UPDATE Command	3-10
3.12	SAVE Command	3-10
3.13	RUN Command	3-11
3.14	BYE Command	3-13

4 — Editing Capabilities

4.1	Introduction	4-1
4.2	Writing Program Lines	4-1
4.2.1	Program Development Mode	4-1
4.2.2	Immediate Execution Mode	4-2
4.3	Using Comments	4-3
4.3.1	Remark Statement (REM)	4-3
4.3.2	Tail Remark	4-4
4.4	Editing Program Lines	4-4
4.5	Adding Lines to a Program	4-4
4.6	Deleting Lines From a Program	4-5
4.7	Copying/Moving Lines in a Program	4-5
4.8	Editing Functions	4-5
4.8.1	Space Forward	4-5
4.8.2	Move Cursor Right/Move Cursor Left	4-5
4.8.3	Back Tab	4-5
4.8.4	Return	4-6
4.8.5	Display Current or Preceding Line	4-6
4.8.6	Display Current or Succeeding Line	4-6
4.8.7	Insert Character	4-6
4.8.8	Delete Character	4-7
4.8.9	Erase Input	4-7
4.8.10	Erase Field	4-7
4.8.11	Tab	4-7
4.8.12	Repeat	4-7
4.8.13	Replay	4-7
4.8.14	Calculate	4-7
4.8.15	Break Execution	4-7
4.8.16	Resume Execution	4-7
4.8.17	Step	4-8
4.8.18	Reset Cursor	4-8
4.8.19	Suspend Execution on Output	4-8
4.8.20	Return with EOF	4-8
4.9	OPTION Statement	4-8
4.9.1	OPTION 1	4-9
4.9.2	OPTION 2	4-10

Paragraph	Title	Page
5 — Data and Expressions		
5.1	Introduction	5-1
5.2	Data	5-1
5.3	Constants	5-1
5.3.1	Numeric Constants	5-2
5.3.2	String Constants	5-3
5.4	Variables	5-3
5.4.1	Numeric Variables	5-4
5.4.2	Numeric Variable Names	5-4
5.4.3	Numeric Type Declarations	5-4
5.4.3.1	The REAL Statement	5-5
5.4.3.2	DECIMAL Statement	5-5
5.4.3.3	INTEGER Statement	5-6
5.4.4	String Variables	5-6
5.4.5	String Variable Names	5-7
5.5	Value Assignment	5-7
5.5.1	Arithmetic LET	5-7
5.5.2	String LET	5-8
5.6	Arrays	5-8
5.6.1	DIM Statement	5-9
5.6.2	OPTION BASE Statement	5-11
5.7	Virtual Arrays	5-12
5.7.1	ASSIGN Statement	5-12
5.7.2	CLOSE Statement	5-14
5.8	Expressions	5-14
5.9	Operators	5-15
5.9.1	Arithmetic Operators	5-15
5.9.2	String Operator	5-15
5.9.3	Relational Operators	5-16
5.9.4	Logical Operators	5-17
5.9.4.1	Logical NOT	5-17
5.9.4.2	Logical OR	5-17
5.9.4.3	Logical AND	5-17
5.9.4.4	Logical Operator Example	5-18
5.10	Precedence of Operator Evaluation	5-18
5.11	Evaluation of Expressions	5-19
5.11.1	Arithmetic Expressions	5-19
5.11.2	Logical Expressions	5-22
5.11.3	String Expressions	5-22
5.11.4	Relational Expressions	5-22

6 — Input/Output (I/O) Statements

6.1	Introduction	6-1
6.2	File Organization	6-1
6.2.1	Sequential	6-1
6.2.2	Relative Record	6-2
6.2.3	KIF	6-2

Paragraph	Title	Page
6.3	OPEN Statement	6-2
6.3.1	OPEN Statement with File Organization Attribute	6-3
6.3.1.1	Opening a Sequential File or Device	6-3
6.3.1.2	Opening a Relative File	6-4
6.3.1.3	Opening a KIF	6-4
6.3.2	File/Device Format Attribute	6-4
6.3.2.1	DISPLAY Format	6-4
6.3.2.2	INTERNAL Format	6-5
6.3.3	File/Device Record Length Attribute	6-6
6.3.3.1	Variable Record Length	6-6
6.3.3.2	Fixed Record Length	6-6
6.3.3.3	Physical Record Length	6-7
6.3.4	File Life	6-7
6.3.5	File Access Attribute	6-7
6.3.5.1	Output Access Mode	6-7
6.3.5.2	Input Access Mode	6-8
6.3.5.3	Update Access Mode	6-8
6.3.5.4	Append Access Mode	6-8
6.4	CLOSE Statement	6-9
6.5	PRINT AND DISPLAY Statements	6-9
6.5.1	Device Output	6-11
6.5.2	Data Separators	6-11
6.5.2.1	Comma	6-12
6.5.2.2	Semicolon	6-12
6.5.2.3	Apostrophe	6-13
6.5.3	Output Options	6-13
6.5.3.1	PRINT with ERASE ALL Option	6-13
6.5.3.2	PRINT with AT Option	6-14
6.5.3.3	PRINT with SIZE Option	6-14
6.5.3.4	PRINT with BELL Option	6-15
6.5.4	PRINT with USING Option	6-15
6.5.5	IMAGE Statement	6-16
6.5.5.1	IMAGE Format Control Characters	6-17
6.5.5.2	Integer Fields	6-18
6.5.5.3	Decimal Fields	6-19
6.5.5.4	Exponential Fields	6-19
6.5.5.5	Alphanumeric Fields	6-20
6.5.5.6	Literal Fields	6-21
6.5.6	File Output	6-21
6.5.6.1	Sequential File Output	6-21
6.5.6.2	Relative Record File Output	6-22
6.5.6.3	KIF Output	6-24
6.5.7	REPRINT Statement	6-25
6.5.8	SCRATCH Statement	6-26
6.6	INPUT AND ACCEPT Statements	6-26
6.6.1	INPUT Statement	6-27
6.6.2	Keyboard Input	6-28
6.6.2.1	INPUT with ERASE ALL Option	6-28
6.6.2.2	INPUT with AT Option	6-29

Paragraph	Title	Page
6.6.2.3	INPUT with SIZE Option	6-29
6.6.2.4	INPUT with BELL Option	6-30
6.6.2.5	Input Prompting	6-30
6.6.2.6	Input Errors	6-31
6.6.3	INPUT Statement in File Access	6-31
6.6.3.1	Sequential File Input	6-31
6.6.3.2	Relative Record File Input	6-32
6.6.3.3	KIF Input	6-33
6.6.4	ACCEPT Statement	6-35
6.7	Program DATA	6-36
6.7.1	DATA Statement	6-36
6.7.2	READ Statement	6-37
6.7.3	RESTORE STATEMENT	6-38
6.7.3.1	RESTORE Data Statement Pointer	6-38
6.7.3.2	File RESTORE	6-39
6.8	PUNCTUATION Statement	6-41
6.9	Shared Files	6-44
6.9.1	Relative Record File Example	6-46
6.9.2	KIF Example	6-46

7 — Control Statements

7.1	Introduction	7-1
7.2	Unconditional Transfer (GOTO)	7-1
7.3	Computed Transfer (ON-GOTO)	7-2
7.4	Conditional Transfer (IF-THEN-ELSE)	7-3
7.5	Repeated Sequences (FOR-TO-STEP-NEXT)	7-5
7.6	GOSUB and RETURN Statements	7-7
7.7	Computed GOSUB Statement	7-9
7.8	ON ERROR/RETURN	7-10
7.9	END Statement	7-12
7.10	STOP Statement	7-12
7.11	Other Methods of Transferring Control	7-12

8 — Intrinsic Functions

8.1	Introduction	8-1
8.2	Mathematical Functions	8-2
8.2.1	Absolute Value Function (ABS)	8-2
8.2.2	Arctangent Function (ATN)	8-3
8.2.3	Cosine Function (COS)	8-3
8.2.4	Exponential Function (EXP)	8-3
8.2.5	Integer Function (INT)	8-4
8.2.6	Natural Logarithm Function (LOG)	8-4
8.2.7	Sign Function (SGN)	8-5
8.2.8	Sine Function (SIN)	8-5

Paragraph	Title	Page
8.2.9	Square Root Function (SQR)	8-6
8.2.10	Tangent Function (TAN)	8-6
8.3	String Functions	8-6
8.3.1	Convert ASCII to Decimal Function (ASC)	8-7
8.3.2	Break Function (BREAK)	8-8
8.3.3	Length Function (LEN)	8-9
8.3.4	Numeric Function (NUMERIC)	8-9
8.3.5	Position Function (POS)	8-10
8.3.6	Repeat Function (RPT\$)	8-11
8.3.7	Match String Function (SPAN)	8-11
8.3.8	Uppercase Function (UPRC\$)	8-12
8.3.9	Value Function (VAL)	8-13
8.3.10	Character Function (CHR\$)	8-14
8.3.11	Segment Function (SEG\$)	8-15
8.3.12	String Function (STR\$)	8-15
8.4	Date and Time Functions	8-16
8.4.1	Date Function (DAT\$)	8-16
8.4.2	Time Function (TIME\$)	8-17
8.5	Miscellaneous Functions	8-18
8.5.1	Random Number Function (RND)	8-18
8.5.2	Randomize Statement (RANDOMIZE)	8-18
8.5.3	Find Available Space Function (FREESPACE)	8-19
8.5.4	Return Number of Characters in Buffer Function (INKEY)	8-19
8.5.5	Return Character Function (INKEY\$)	8-19
8.5.6	End-of-File Function (EOF)	8-20
8.5.7	Verify File Function (FTYPE)	8-22
8.5.8	Tab Function (TAB)	8-23
8.5.9	ERR Function	8-24
8.5.10	Test for Duplicate Keys (DUP)	8-26

9 — User-Defined Procedures

9.1	Introduction	9-1
9.2	Function Definition	9-1
9.2.1	Define Statement (DEF)	9-2
9.2.2	Function End Statement (FNEND)	9-4
9.2.3	Recursive Functions	9-4
9.3	BASIC Subprograms	9-5
9.3.1	Calling Subprograms	9-5
9.3.2	Subprogram Statements	9-6
9.3.2.1	SUB Statement	9-6
9.3.2.2	ESUB Statement	9-7
9.3.2.3	SUBEXIT Statement	9-8
9.3.2.4	SUBEND Statement	9-8
9.4	Defining Types of Parameters and Local Variables	9-8

Paragraph	Title	Page
-----------	-------	------

10 — Debug Features

10.1	Introduction	10-1
10.2	Program Breaks	10-1
10.2.1	BRKPNT and UNBRKPNT Commands	10-1
10.2.2	Break Function Key	10-2
10.3	Continuing Execution	10-2
10.4	Stepping a Program	10-3
10.5	TRACE and UNTRACE	10-3

11 — Assembly Language Subroutines

11.1	Introduction	11-1
11.2	Creating an Assembly Language Subroutine	11-1
11.3	Installing an Assembly Language Subroutine	11-2
11.4	LIBRARY Statement	11-2
11.5	CALL Statement	11-3
11.6	Parameter Information Block	11-3
11.7	Accessing Parameter Data Values	11-5
11.7.1	Integer Data Format	11-5
11.7.2	Real Data Format	11-5
11.7.3	Decimal Data Format	11-6
11.7.4	String Data Format	11-7
11.8	Accessing Arrays in an Assembly Language Subroutine	11-8
11.8.1	Accessing Numeric Arrays	11-8
11.8.2	Accessing String Arrays	11-9
11.9	Assembly Language Subroutine Example	11-9
11.9.1	Step 1 — Creating the Assembly Language Source Code	11-9
11.9.2	Assembling the Source File	11-13
11.9.3	Step 3 — Linking the Assembly Language Subroutine	11-13
11.9.4	Step 4 — Calling the Subroutine from a BASIC Program	11-13

12 — BASIC Subroutine Library

12.1	Introduction	12-1
12.2	Using the Subroutines	12-1
12.2.1	Subroutine Arguments	12-1
12.2.2	Subroutine Error Codes	12-2
12.3	SORT Subroutines	12-3
12.3.1	Sort Example	12-6
12.3.1.1	Sorted Record Output	12-7
12.3.1.2	Integer Output	12-8
12.4	Keyed File Package	12-9
12.4.1	Keyed File Organization	12-9
12.4.2	Keyed File Format	12-9

Paragraph	Title	Page
12.4.3	Keyed File Data Base Buffer Creation	12-10
12.4.4	Keyed File Creation	12-10
12.4.5	KFP Memory Management	12-10
12.4.5.1	Subroutine Memory Requirements	12-10
12.4.5.2	Keyed File Data Base Buffer	12-11
12.4.5.3	Record Buffer	12-12
12.4.6	KFP Subroutines	12-12
12.4.6.1	BLDBUF — Build Keyed File Data Base Buffer File.....	12-13
12.4.6.2	KFINIT — Initialize Keyed File Data Base Buffer	12-13
12.4.6.3	KFCREA — Create Keyed File	12-15
12.4.6.4	KFOPEN — Open Keyed File	12-16
12.4.6.5	KFPUT — Put Data into Record Buffer	12-17
12.4.6.6	KFWRITE — Write Keyed File	12-19
12.4.6.7	KFREAD — Read Keyed File	12-20
12.4.6.8	KFGET — Get Data from Record Buffer	12-21
12.4.6.9	KFDEL — Delete Keyed File Record	12-22
12.4.6.10	KFCLOS — Close Keyed File	12-23
12.4.7	Keyed File Example	12-23

Appendixes

Appendix	Title	Page
A	Keycap Name Cross-Reference	A-1
B	ASCII Character Set	B-1
C	BASIC Reserved Word List	C-1
D	Error Messages and Codes	D-1
E	BASIC System Differences	E-1
F	Logical Operators with Integer Operands	F-1
G	Syntax Diagrams	G-1
H	Example Programs	H-1
I	BASIC Systems Function Keys	I-1

Index

Illustrations

Figure	Title	Page
2-1	Invader	2-4
7-1	Nested Subroutines	7-8
11-1	Parameter Information Block	11-4
11-2	Integer Data Format	11-5
11-3	Real Data Format	11-5
11-4	String Data Format	11-7
11-5	Sample Routine Source Code	11-10
11-6	Sample Routine Object Listing	11-14
11-7	Sample Routine Object Listing Cross-Reference	11-19
11-8	Link Control File for Sample Routine	11-20
12-1	Sort Subroutine Program	12-7
12-2	Sample Build Buffer Utility	12-25
12-3	Keyed File Example	12-26

Tables

Table	Title	Page
1-1	TI BASIC Device Support	1-2
4-1	Bit Positions and Values Associated with Keys	4-9
5-1	Arithmetic Operators	5-15
5-2	Relational Operators	5-16
5-3	Truth Table	5-17
5-4	Numeric/Logical Expression Priority	5-18
6-1	Memory Requirements for INTERNAL Format Data	6-5
6-2	Format Control Characters	6-17
8-1	BASIC Intrinsic Functions	8-1
8-2	FTYPE Values	8-22

General Description

1.1 INTRODUCTION

The BASIC system is a commercially oriented program development system. The BASIC language includes the minimal BASIC standard developed by the American National Standards Institute (ANSI), designated ANSI X3.6-1978. Texas Instruments has developed BASIC beyond this standard, producing a more flexible language system suitable for both business and scientific applications. By using the extensions made to TI BASIC, you can perform the following:

- Write independent subroutines in BASIC
- Write multiline function definitions
- Specify images that provide precise control of output formats
- Accommodate arrays too large for internal memory
- Transmit data to and from files and external devices
- Specify the position of data on the screen
- Restrict the precision of data by rounding
- Conveniently locate program logic errors
- Determine the date and time
- Use IF-THEN-ELSE statements
- Use assembly language subroutines within BASIC programs
- Protect programs against modification and examination
- Write multiple statements on each program line

1.2 BASIC SYSTEM CONFIGURATION

You can install TI BASIC on Business System 200 (S200) computers that use the DX10 Micro (DXM) operating system, or on other Business System or TI 990 computers that use the DX10 or DNOS operating system. The capabilities of BASIC are identical on the various systems, with the exception that DXM is a single-user, single-task operating system. While DX10 and DNOS can execute separate tasks in foreground and background mode, DXM executes one task at a time in foreground mode. BASIC executes as a single task; thus, on S200 computers, when you are developing or running a program in BASIC, the System Command Interpreter (SCI) suspends operation. Appendix E discusses system differences.

Standard configuration for a TI BASIC system includes at least one system and one terminal device; the capacity to support additional devices depends on the operating system. The terminal device is the device through which you enter BASIC commands. The system device (usually a disk) is the external storage unit for any BASIC operating system that does not reside in the computer's internal memory. A file device is any device that can be attached to the computer with which a BASIC program communicates. Table 1-1 lists the equipment that can be used as devices with each of the systems.

Table 1-1. TI BASIC Device Support

BASIC OS	Terminal Device	System Device¹	File Device²
DXM	any	DSDD or Winchester	DSDD or Winchester
DX10	any	any disk ³	any
DNOS	any	any disk ³	any

Notes:

¹ DSDD indicates double-sided, double-density diskettes.

² You can use any device for a file, but you must provide a device service routine (DSR) for it. The standard release packages include DSRs for the terminal device, the system device, and printers available with the operating system.

³ The devices that can be used in these positions are restricted only by the DX10 or DNOS operating system.

1.3 MANUAL OVERVIEW

This manual is arranged in top-down order. Section 2 tells you how to get into and out of BASIC; we work down from there. The following list briefly describes the elements of TI BASIC in the order they appear in this manual:

- **Commands** — Section 3 describes the BASIC commands that allow you to instruct the computer to perform immediate actions.
- **Editing operations** — Section 4 describes TI BASIC's extensive editing capabilities, which facilitate creating and editing program statements.
- **Data and expressions** — Section 5 describes how TI BASIC handles data and explains how to assign values in program statements.
- **Statements** — Allow you to construct a program. Each statement causes the program to perform a particular function. Section 6 describes I/O statements; Section 7 describes control statements (for example, GOTO).
- **Intrinsic Functions** — Section 8 describes the intrinsic functions provided by TI BASIC. They include mathematical functions, string functions, date and time functions, and miscellaneous functions that perform commonly used operations. These functions are accessed by statements.
- **User-defined functions and subroutines** — Section 9 describes how to write your own functions and subroutines. You can minimize code by writing a function or subroutine that is activated at several locations within a program but written only once.
- **Debugging Aids** — Section 10 describes the BASIC debugging features that allow you to examine the actions of the program and identify functional errors.
- **Assembly-language subroutines** — Section 11 outlines how to create, link, and call assembly language subroutines to interface with your program. These routines enable you to implement special functions and features not supplied in BASIC.
- **Error Messages** — Appendix D lists the error messages and codes. The error message indicates the nature of the error and where it was detected in the program.

1.4 PROGRAM DEVELOPMENT

The following paragraphs outline the steps in program development.

1.4.1 Writing the BASIC Program

A BASIC program is made up of one or more BASIC statements, which may be entered in either of two modes:

- Program development mode — In this mode, the programmer enters a line number followed by a BASIC statement. The statements do not execute until the program executes.
- Immediate execution mode — In this mode, BASIC statements execute as soon as you enter a carriage return. There are no line numbers.

Each BASIC statement has a required syntax. To terminate a statement, enter any editing function that causes a carriage return. You can write multiple statements on the same line by using a double colon (::) to separate the statements.

1.4.2 Listing the Program

When a new program has been created or an existing program has been loaded, you can list the program. Listing the program allows you to check the statements entered.

1.4.3 Executing the Program

After a program has been written, it can be executed. When a program error is encountered, the BASIC system displays an error message and stops execution or transfers control to a user-supplied error routine. Appendix D describes the system-supplied error messages. The reappearance of the prompting period indicates successful completion of the program.

1.4.4 Editing the Program

When you discover errors in a program or need to add or change statements, you can edit the program to make the appropriate changes. The BASIC editing functions allow you to display and edit the lines of a program and to add new lines.

1.4.5 Correcting the Program

You can locate and correct most errors by using the information in the error message descriptions in Appendix D. Each error message includes a numeric code. Appendix D lists the error messages in numeric order according to the code. If you cannot correct an error, use the debugging commands discussed in Section 10 to locate the error.

1.4.6 Saving the Program

When you complete work on a program or decide to continue the work later, you can save the program in a file. To resume work or execute the program, use a BASIC command and the pathname of the file.

Getting Started

2.1 INITIAL OPERATIONS

To begin to develop or execute a BASIC program, enter the SCI procedure name BASIC. One set of the following prompts appear:

DXM only

EXECUTE BASIC	EXECUTE BASIC
WORKSPACE SIZE (KB) :	WORKSPACE SIZE (KB) : 20
NUMBER OF OVERLAY BUFFERS: 8	NUMBER OF OVERLAY BUFFERS: 6
WORK FILE VOLUME NAME:	WORK FILE VOLUME NAME:
INITIAL PROGRAM NAME:	INITIAL PROGRAM NAME:
MODE (F,B) : FOREGROUND	

Prompt Responses

WORKSPACE SIZE

Allows you to specify, in increments of 1000 bytes (or *kilobytes*), the amount of available user memory you want to use for program editing and execution. The system accepts integer values; for example, if you want 12,000 bytes of memory available, you would enter 12 in response to this prompt. There is no default value. If you request more memory than is available, the system assigns all available memory.

NUMBER OF OVERLAY BUFFERS

Allows you to specify the number of overlay buffers available to this BASIC program. The more buffers you assign, the less likely BASIC is to have to access an overlay on disk; however, each buffer takes up memory that otherwise would be available to the user. You should assign the lowest value that allows your program to run efficiently. Experiment with each application to determine the optimum response to this prompt.

WORK FILE VOLUME NAME

The name of the volume where the BASIC work file resides. Although no default is indicated, the default value is the system disk.

INITIAL PROGRAM NAME

Indicates that you have the option of loading and executing a BASIC program immediately. If you enter a valid program name, that program executes immediately.

MODE

Requests whether BASIC is to be executed in background or foreground. On operating systems that have background mode, if you select this mode, you must specify an initial program to run in background. The program cannot contain DISPLAY, ACCEPT, INPUT, or PRINT statements that direct I/O to your terminal. If the program contains an error, the system displays the appropriate error message; otherwise, on termination the system displays a message indicating successful completion. If you select foreground mode, you can execute a program immediately or enter the BASIC command mode.

If you enter an invalid name or press the Return key in response to the INITIAL PROGRAM NAME prompt, the system displays the following information:

```
TEXAS INSTRUMENTS BASIC VERSION n.n.n
```

The prompting period appears, indicating that you are in command mode. You can now enter any of the BASIC commands (see Section 3).

The first time you execute BASIC, no initial value appears for the WORKSPACE SIZE(KB) prompt. However, after you enter a value for this prompt, that value appears as the initial value each time you execute BASIC until you specify a new response. Similarly, if you enter a program pathname in response to the prompt, that pathname appears the next time you execute BASIC.

The existence of a work file allows you to write and execute a program that, including its subprograms, is larger than available user memory. The work file contains the main program and external BASIC subprograms. When you are ready to edit the main program or a BASIC subprogram, you enter the appropriate edit command (see Section 4). That routine is copied into memory for editing. Only one routine is in memory at a time.

To terminate BASIC, enter the command BYE after the prompting period.

2.2 SAMPLE PROGRAM

This paragraph provides a sample program for users who prefer to get acquainted with their systems through hands-on experience. The following steps give you a brief introduction to program development in TI BASIC. You will:

1. Enter BASIC in the program development mode
2. Build a program source module
3. Execute the program
4. Terminate BASIC

2.2.1 Step 1 — Enter BASIC

To begin creating your program, press the Command key to place SCI in the command mode. Now, enter the BASIC command by typing BASIC following the SCI prompt and press Return. Respond to the prompts as follows:

```
EXECUTE BASIC
      WORKSPACE SIZE (KB): 4
NUMBER OF OVERLAY BUFFERS: 8
      WORK FILE VOLUME NAME:
      INITIAL PROGRAM NAME:
      MODE (F,B): FOREGROUND
```

You are in the program development mode; you can begin to enter source code.

2.2.2 Step 2 — Create the Program

Step 2 explains how to write a BASIC program source module using the Text Editor. For a complete description of the Text Editor, see Section 3.

1. The prompting period indicates that BASIC is in the command mode. Enter the NEW command, which erases any program currently in memory, as follows:

```
NEW
```

2. To eliminate the need to type in line numbers, enter the NUM command. The system will start numbering at 100, using increments of 10.

```
NUM
```

3. Type in the following program. By pressing the Return key, you terminate a program line. Figure 2-1 offers you an entertaining introduction to TI BASIC. Refer to Step 3 (paragraph 2.2.3) for an analysis of the coding techniques employed in this program.
4. Check your program for typos. If you discover a mistake, press the Previous Line or Next Line key until you reach the line number of the incorrect statement. Simply write over it and press Return; the new line replaces the incorrect one.
5. To disable the automatic numbering initiated by the NUM command, press Return when line 1110 appears on the screen. Use the SAVE command followed by a pathname and LIST to save this program on disk. Type in the command SAVE immediately after the prompting period; since this command is not part of the program, you do not want it to have a line number. Enter the following line to save your program:

```
SAVE ".INVADER" LIST
```

The name of the file is INVADER; the period preceding the file name indicates that the file is on the system disk. The LIST parameter specifies that the file is saved in source form. If you omitted LIST, the program would be saved in object code; you could run it, but could not correct errors.

```

100 INTEGER ALL::RANDOMIZE
110 DISPLAY ERASE ALL
120 DISPLAY AT(4,34) " INVADER "
130 DISPLAY AT(8,21) " USE THE FOUR AND SIX NUMERIC KEYS "
140 DISPLAY AT(9,21) " TO CONTROL THE DIRECTION OF THE "
150 DISPLAY AT(10,21) " LASAR CANNON MOVING ALONG THE "
160 DISPLAY AT(11,21) " BOTTOM OF THE SCREEN. THE FIVE "
170 DISPLAY AT(12,21) " KEY WILL STOP THE CANNON AND THE "
180 DISPLAY AT(13,21) " SPACE BAR FIRES A LASAR BLAST. "
190 DISPLAY AT(14,21) " THE ESC KEY ABORTS THE CURRENT "
200 DISPLAY AT(15,21) " MISSION. PRESS RETURN TO BEGIN. "
210 DISPLAY AT(24,1)"";
220 A$=INKEY$(0)::IF ASC(A$)<>141 THEN 220 !Wait for RETURN key
230 HC=0
240 DISPLAY ERASE ALL AT(1,1)"POINTS: ";HC :: THITS=0
250 RKT=25 :: DISPLAY AT(2,1)"SHOTS LEFT:";RKT
260 CUR=40 :: DLY=0 :: STP=53:: LFT=52 :: RIT=54 :: ESC=155::FIR=32
270 TRW=4 :: SPD=2 :: DIR=0 :: BWAT=70 :: LBR=5 :: LBC=1 !Sets up keys,speed,etc
280 BMB=0 :: BCLK=0 :: TOG=-1
290 T$=CHR$(27+128)&"*"&CHR$(11+128) !Defines target
300 !
310 TGT=INT(75*RND)+1 :: LTRW=TRW :: TRW=INT(17*RND)+4
320 DISPLAY AT(LTRW,1) :: DISPLAY AT(TRW,TGT)T$ !Displays target
330 !
340 HOLD=INT((80-(PCNT/2))*RND) !How long to hold still
350 FOR TIM= 1 TO HOLD !Begin loop -----
360 IF BMB THEN BWAT=INT(50*RND) !
370 IF NOT BMB THEN BCLK=BCLK+1 !
380 IF NOT BMB THEN IF BCLK>BWAT THEN BMB=-1::BRW=TRW::BCL=TGT+2::BCLK=0 !
390 DISPLAY AT(23,LCUR) " " !
400 DISPLAY AT(23,CUR) CHR$(28) !
410 TOG=TOG*-1 :: IF TOG=-1 THEN DISPLAY AT(24,1) " " !
420 IF BMB THEN BMB=MOVB :: IF BMB>0 THEN 490 !
430 ! !
440 IF INKEY(0)=0 THEN 570 !
450 M$=INKEY$(0) :: BTN=ASC(M$) !Accept,decode command!
460 IF BTN=FIR THEN TIM= FIRE(CUR,TGT) ELSE IF BTN=LFT THEN DIR=-SPD !
470 IF BTN=RIT THEN DIR=SPD ELSE IF BTN=STP THEN DIR=0 !
480 IF BTN<>ESC AND RKT<>0 THEN 570 !
490 ! !
500 IF RKT=25 THEN PCNT=0 ELSE PCNT=(THITS/(25-RKT))*100 !
510 DISPLAY AT(1,20)"PERCENT: "&STR$(PCNT)&"%" !
520 ACCEPT AT(24,1)SIZE(1),"AGAIN(Y/N)?: ":A$ !
530 IF A$="Y" OR A$="" THEN 240 !Start up again !
540 ! !
550 IF A$<>"N" THEN 530 ELSE STOP !Quit !
560 ! !
570 IF CUR+DIR>80 OR CUR+DIR<1 THEN DIR=-DIR !Cursor wraparound !
580 ! !
590 LCUR=CUR :: CUR=CUR+DIR !
600 NEXT TIM !End loop -----
610 GOTO 310
620 !
630 DEF FIRE(SRC,DST) !Fires at target
640 FIRE=HOLD !:: DIR=0
650 RAY$=CHR$(9) !Defines rocket path
660 FOR Z=22 TO TRW STEP -1 !Shoot rocket at target
670 DISPLAY AT(Z,SRC)SIZE(1)RAY$ !Shows path
680 IF Z=BRW AND SRC=BCL AND ASC(RAY$)=9 THEN BMB=0::HC=HC+1
690 NEXT Z
700 IF RAY$=CHR$(9) THEN RAY$=" " :: GOTO 660

```

Figure 2-1. Invader (Sheet 1 of 2)

```

710         IF SRC<DST OR SRC>DST+(LEN(T$)-1) THEN 730 !It's a hit!
720         HC=HC+2::THITS=THITS+1::DISPLAY AT(1,1)CHR$(7)::CALL EXPL(TRW,TGT+1,0)
730         DISPLAY AT(1,12) HC
740         RKT=RKT-1                                     !One down and
750         DISPLAY AT(2,12) RKT                         !How many to go
760         IF RKT=25 THEN PCNT=0 ELSE PCNT=(THITS/(25-RKT))*100
770 FNEND
780 !
790 DEF MOVB
800     BRW=BRW+1
810     B$="+"
820     IF BCL<CUR THEN BM=1 ELSE BM=(BCL>CUR)
830     IF ABS(BCL-CUR) >23-BRW THEN BM=BM*2
840     BCL=BCL+BM
850     DISPLAY AT(LBR,LBC) SIZE(1) " " :: LBR=BRW :: LBC=BCL
860     DISPLAY AT(BRW,BCL) SIZE(1) B$
870     IF (BCL>=TGT AND BCL<=TGT+(LEN(T$)+1)) AND BRW=TRW THEN TIM=HOLD
880     IF BRW<23 THEN 980
890     IF BCL<>CUR THEN 950 ELSE DISPLAY AT(1,1)CHR$(7);
900     FOR Z=1 TO 50
910         DISPLAY AT(23,CUR-5) "_\`|^/_"
920     NEXT Z
930     DISPLAY AT(23,1)
940     BMB=1::GOTO 980
950     DISPLAY AT(LBR,LBC-2) SIZE(5) ""
960     DISPLAY AT(24,LBC-2) SIZE(5) "\.|./"
970     BMB=0
980     MOVB=BMB
990 FNEND
1000 ! SHOW AN EXPLOSION
1010 SUB INTEGER EXPL(R,C,T)
1020 IF T OR R<6 OR R>20 OR C<5 OR C>75 THEN N=1 ELSE N=3
1030 EC$=CHR$(19)
1040 K=R :: L=C :: PRINT AT (K,L) " ";
1050 FOR J=1 TO N :: DISPLAY AT(K,L-J);EC$; :: DISPLAY AT(K-J,L-J) EC$;
1060 DISPLAY AT(K-J,L) EC$; :: DISPLAY AT(K-J,L+J) EC$;
1070 DISPLAY AT(K,L+J) EC$; :: DISPLAY AT(K+J,L+J) EC$;
1080 DISPLAY AT(K+J,L) EC$; :: DISPLAY AT(K+J,L-J) EC$;::NEXT J
1090 IF EC$<>" " THEN EC$=" " ::GOTO 1050
1100 SUBEND

```

Figure 2-1. Invader (Sheet 2 of 2)

2.2.3 Step 3 — Analyze the Coding

This program illustrates many of the coding techniques described in later sections of the manual. If you wish, read the following commentary to become acquainted with some of the features of TI BASIC. You may want to study the coding in the program more closely after reading the rest of the manual.

LINE 100

The INTEGER ALL statement assigns the integer data type to all variables in the program.

LINE 110

DISPLAY ERASE ALL clears the screen.

LINES 120 through 210

These DISPLAY statements display instructions on the screen, explaining to the player how to move and fire the cannon.

LINE 220

Waits for the player to press Return.

LINES 230 through 280

These lines clear the instructions from the screen, then display the number of shots available to the player. Constants are assigned values (for example, STP (Stop) is assigned the value of 53; SPD (Speed) equals 2).

LINE 290

Defines the target, the spaceship.

LINES 300 through 330

Displays the target.

LINE 340

Defines how long the target will hold still. The better the player is, the shorter the time the spaceship remains in one place. The percentage of hits is divided by 2, subtracted from 80, and then multiplied by a random number.

LINES 350 through 620

The main body of the program times the bombs falling from the spaceship, displays the cannon, defines what to do when the cannon is hit, etc.

LINES 630 through 780

A function the programmer has defined; this function controls the cannon fire. See Section 9 on user-defined functions.

LINES 790 through 990

Another user-defined function; the purpose of this one is to move the bomb. Line 810 defines the bomb as the “+” character. The next two lines make the bombs fall toward the cannon. Line 870 prevents the bomb from hitting the spaceship; lines 900 through 970 define what is displayed for a hit and a miss.

LINES 1000 through 1100

A subprogram that shows an explosion.

2.2.4 Step 4 — Execute the Program

Enter the RUN command to execute the program. You would enter the following line to run .INVADER:

```
RUN ".INVADER"
```

The program should execute correctly. If it does not, error messages will appear indicating the line number and kind of error. Check the line indicated against the line in the source program and correct it. The program still resides in memory; to make corrections, enter the LIST command. Then use the Previous Line to find the incorrect line or lines. For more information on BASIC error messages and codes, refer to Appendix D. For information on the BASIC commands, see Section 3. For information on using the BASIC editing capabilities, see Section 4.

2.2.5 Step 5 — Terminate BASIC

When you have saved the world from space invaders and are ready to terminate BASIC, enter the command BYE after the prompting period, as follows:

```
BYE
```

BYE terminates BASIC and the system returns to SCI mode.

BASIC Commands

3.1 INTRODUCTION

Commands direct and control system functions, including creating programs, debugging programs, and terminating system operations. This section discusses program development commands and the BYE command, which terminates operations. Section 10 discusses debug commands. You can enter commands whenever the prompting period (.) is displayed. Some commands (such as BYE, RUN, RENAME, and DELETE File) are executable as either BASIC commands or statements.

Each command is associated with a keyword. When you use a command as a statement in a BASIC program, only the first three letters of the keyword need be entered. Some commands require that you enclose the pathname of a BASIC program in quotes. For example, when loading the program in Section 2, you would use OLD “.INVADER”.

When you enter BASIC, you can choose among three operations:

- Develop a new program
- List or modify an old program
- Run a program

The BASIC commands in the following paragraphs are described as you might use them in the three operations. If you were developing a new program, you would begin with NEW, then probably number the lines (NUM), and later might want to resequence them (RES).

If you were editing an old program, you would begin with OLD. You could then specify which portion of the program to edit, the main program or a subroutine (EDIT). Once the portion to edit is loaded into memory, you might list it to the screen or optional printer (LIST). The DELETE commands allow you to delete extraneous lines or files. The RENAME command allows you to rename a file. The MERGE command allows you to merge two programs. When you are debugging a program, you can use the UPDATE command to send the modifications you have completed to the work file without affecting the program on disk. That way, you can experiment with modifications. When you finish debugging a program, you are ready to save it.

The SAVE command allows you to store a program in source format (SAVE LIST), in internal (object) format (SAVE), or in object format with program security (SAVE LOCK).

The RUN command runs a program saved in source or object format. Programs saved in object format run faster but cannot be debugged or edited.

3.2 NEW COMMAND

The NEW command prepares the system to accept a new program by erasing the program currently in memory and emptying the BASIC work file. BASIC is in the immediate execution mode. This command has no parameters. NEW removes all program lines and variables currently in memory, clears the BASIC work file, closes any open files, and deactivates the TRACE command (Section 10).

FORMAT

NEW

NEW SIZE workspace

NEW SIZE workspace, overlays

where:

workspace is the size, in kilobytes, of the workspace you wish to define for the new program.

overlays is the number of overlay buffers you want available for this application.

3.3 NUMBER COMMAND

The NUMBER command eliminates the need to type line numbers when entering a program from the keyboard. Line numbers in BASIC never contain commas.

FORMAT

NUMBER

NUMBER line__num

NUMBER ,inc

NUMBER line__num,inc

where:

line__num is the beginning line number.

inc is the increment used to determine subsequent line numbers.

If you do not specify a starting line number, the system begins with a default line number of 100. If you do not specify an increment, the system uses a default increment of 10. If the system reaches a line number that currently exists in program memory, the existing line and its line number appear on the screen.

The following is an example of the NUMBER command.

```
NUMBER 1100,50
```

This command generates line numbers for statements, beginning with line number 1100. The system numbers lines by increments of 50 until a blank line is entered. If 50 were omitted, the system would use the default increment of 10.

3.4 RESEQUENCE COMMAND

The RESEQUENCE command renumbers the program lines currently in memory.

FORMAT

```
RESEQUENCE
```

```
RESEQUENCE line__num
```

```
RESEQUENCE line__num, inc
```

```
RESEQUENCE line__num, inc, start__line - end__line
```

where:

line__num is the beginning line number.

inc is the increment used to determine subsequent line numbers.

start__line is the first original line number to be changed.

end__line is the last original line number to be changed.

You can omit any parameter, in which case the following defaults apply:

- First new line number: 100
- New line number increment: 10
- First original line number to be changed: 1
- Last original line number to be changed: 32759

Note that the defaults for the first two parameters are the same as for the NUMBER command.

The RESEQUENCE command assigns new line numbers to all lines in the specified range. The command also changes all line references in BASIC statements contained within memory to match the new numbers. The RESEQUENCE command cannot change the order of program lines in memory.

The following example illustrates the most general form of the command:

```
RESEQUENCE 4000,20,10000-12000
```

The parameters in the example specify the following:

- 4000 — First new line number to be used when renumbering
- 20 — Line number increment for the renumbering
- 10000 — First original line number to be changed
- 12000 — Last original line number to be changed

This command is particularly useful when you are developing a program and require more space to insert additional statements. After you enter RES and accept the defaults, the entire contents of memory are renumbered, statements that reference line numbers are modified (for example, GOTO), and space for nine new statements is provided between each existing statement.

You can also use the RESEQUENCE command in conjunction with the MERGE command when appending several subroutines to a program in memory. If you are using the convention of saving all subroutines resequenced to 20000, 10, or some comparably large range of numbers, you need to ensure that you do not duplicate line numbers. You can use the RESEQUENCE command to renumber the program each time a MERGE command brings in a subroutine.

3.5 OLD COMMAND

The OLD command copies the main program and external subprograms from disk files into the BASIC work file. OLD then loads the main program into the user workspace, erasing the program previously residing in memory there. External subprograms remain in the BASIC work file rather than being loaded into memory.

FORMAT

OLD pathname

OLD pathname SIZE workspace

OLD pathname SIZE workspace, overlays

where:

pathname is a valid pathname.

workspace is the size, in kilobytes, of the workspace you wish to define for this program.

overlays is the number of overlay buffers you want available for this application.

Enclose the pathname of the file containing the program in quotes. When loading is complete, the system returns to the command level.

3.6 EDIT COMMAND

The EDIT command copies the main program or an external subprogram from the BASIC work file into memory for editing.

3.6.1 EDIT MAIN

This form of the EDIT command loads the source main program from the BASIC work file into the user workspace for editing.

FORMAT

EDIT MAIN

The main program currently in the workfile is loaded into memory. If no program is in the BASIC work file, an error message appears.

3.6.2 EDIT ESUB

This form of the EDIT command loads a disk-resident subprogram from the BASIC work file into the user workspace for editing.

FORMAT

EDIT ESUB name

where:

name is the name of the external subprogram.

The external subprogram specified by name is loaded into memory. If the specified subprogram is not in the BASIC work file, an error message appears.

3.7 LIST COMMAND

The LIST command lists on the screen the specified line(s) of the program or subprogram currently in memory. You cannot use this command within a program.

FORMAT

LIST

LIST start__line__num - end__line__num

LIST line__num, line__num

where:

start__line__num specifies the first line to be listed.

end__line__num specifies the last line to be listed.

line__num is an individual number to be listed.

The following are examples of the LIST command:

LIST	Displays the entire program contained in memory
LIST -30	Displays lines 30 and below
LIST 30	Displays line 30
LIST 120-	Displays lines 120 and above
LIST 500-1000	Displays lines 500 through 1000
LIST 10-50, 100-200	Displays lines 10 through 50 and 100 through 200

To abort a listing, use the break execution function (Section 4). Paragraph 3.12 discusses listing to devices other than the screen.

3.8 DELETE COMMAND

The DELETE command deletes lines from a program or deletes a file.

3.8.1 DELETE File

This form of the DELETE command deletes a file.

FORMAT

DELETE pathname

The file specified by pathname is deleted. If the file does not exist, BASIC ignores the command.

3.8.2 DELETE ESUB

This form of the DELETE command deletes a subprogram.

FORMAT

DELETE ESUB name

where:

name is the name of the external subprogram

The subprogram specified by name is deleted. If the subprogram does not exist, an error message appears.

3.8.3 DELETE Lines

This command deletes lines from the program currently in memory. You cannot use this command during program execution.

FORMAT

DELETE start__line__num - end__line__num

DELETE line__num,line__num

where:

start__line__num specifies the first line to be deleted.

end__line__num specifies the last line to be deleted.

line__num is an individual number to be deleted.

The following is an example of the DELETE command.

EXAMPLE

DELETE 120-200

In the example, line numbers 120 through 200 (inclusive) are deleted. If you omit either the starting or the ending line number, all preceding or succeeding lines, respectively, are deleted. If you specify only a single line number without a leading or trailing dash, only that line is deleted. Another way to delete a single line is to enter the line number and press Return. The following example shows multiple ranges being deleted:

DELETE -20,70-120,400-

The command in this example deletes lines 1 through 20, lines 70 through 120, and lines 400 and above.

3.9 RENAME COMMAND

The RENAME command changes the name of a file.

FORMAT

```
RENAME pathname__1 TO pathname__2
```

The current pathname, pathname__1, is renamed pathname__2. If the file to be renamed does not exist or a file with the new name already exists, an error message appears and no action is taken. Both pathnames must specify the same disk. The file being renamed must be closed before the RENAME command executes. The command is executable from a program.

3.10 MERGE COMMAND

You can use the MERGE command to merge programs saved in SAVE and SAVE LIST format; you cannot merge programs saved in SAVE LOCK format.

The MERGE command merges a program on a disk file with the program or external subprogram currently in memory. This command is especially useful when used with subprograms. The program text in the named file is added to the text already in memory.

FORMAT

```
MERGE pathname
```

The file specified by pathname is the file to be merged with the program in memory. If any line in the program on disk has the same number as the program in memory, the former replaces the latter in memory. To avoid interspersing lines of the program on the disk file with those of the program in memory, ensure that the program on disk has higher line numbers than the program in memory.

3.11 UPDATE COMMAND

The UPDATE command writes the contents of the user workspace (the main program or external subprogram (ESUB) in memory) to the BASIC work file. After you execute the UPDATE command, you can bring new material into the workspace to edit without losing the corrections made to the previous material. However, note that edit changes do not become permanent until you enter the SAVE command.

FORMAT

UPDATE

The UPDATE command is useful for editing programs containing external subprograms. The work file is a temporary file; modifications made to its contents do not affect the original program. You can experiment with modifications to a program, run it, and debug it again if necessary. When you finish editing the program, you can make the modifications permanent by entering the SAVE command.

3.12 SAVE COMMAND

The SAVE command saves the program in memory as a program on a disk file or lists it to a printer.

FORMAT

SAVE pathname LIST

SAVE pathname

SAVE pathname LOCK

In all three formats, SAVE writes the contents of the user workspace (the program or subprogram in memory) to the BASIC work file. The contents of the BASIC work file are then copied to the disk file specified by pathname.

In the first format, the LIST parameter directs the system to translate the program in the work file to readable ASCII text before copying it to the sequential file designated by pathname. You must store a program in this format if it is to be read by another computer. At least one copy of every program should be saved with the LIST option to ensure compatibility with future BASIC releases and for file compatibility between BASIC systems.

The second format omits the LIST specification and saves the program in internal (object) format as a relative record file. Although this format allows quicker loading, you cannot look at the file using the Show File (SF) or Print File (PF) commands.

In the third format, the LOCK parameter directs the system to save the program in protected format (or Fast). The program in the BASIC work file is translated to Fast format before it is copied to the file on disk designated by pathname. The header record of this file is marked as Fast format. Large programs saved in this format usually execute considerably faster than programs saved without LOCK.

You cannot list or load programs saved with LOCK. This feature allows you to protect the program from being modified. A program is saved in locked format only if you specify the LOCK parameter.

CAUTION

Before saving a program in LOCK format, be sure you have another copy of the program. You cannot unlock a program saved in LOCK format.

To list a program on a printer, specify the printer in the SAVE command and specify the LIST option, as shown in the following example:

EXAMPLE

```
SAVE "LP01" LIST
```

This command lists on the printer the program currently in memory. Note that the LIST option must be specified.

3.13 RUN COMMAND

The RUN command executes the program in memory or a program saved in a file. If the program has been saved to a file, the contents of the file are copied to the BASIC work file and execution begins. Programs saved in SAVE or SAVE LIST format execute more slowly than programs saved in SAVE LOCK format.

FORMAT

RUN

RUN line__num

RUN pathname

RUN pathname line__num

RUN pathname line__num SIZE

RUN pathname line__num SIZE workspace

RUN pathname line__num SIZE workspace, overlays

where:

line__num specifies the line where program execution begins.

pathname specifies the file pathname of the program to be executed.

workspace is the size of the workspace you wish to define for this program.

overlays is the number of overlay buffers you want available for this application.

The following shows several examples of the RUN command:

RUN

RUN 570

RUN "DS02.MYFILE"

RUN "VOL1.MYFILE" 120

Execution begins with the lowest-numbered line unless you specify a line number. Executing any form of the RUN statement destroys the values of all variables in memory.

The first example executes the program in memory from the beginning, and the second example starts execution at line number 570. The third example loads the program DS02.MYFILE from disk and executes it from the beginning. Any program currently in memory and all variable values are lost. This command can be executed in one program to chain it to another program. No variable values can be passed to the succeeding program except through virtual arrays or files. In the last example, VOL1.MYFILE is loaded and execution begins at line number 120. If the line number specified does not exist, an error message appears.

3.14 BYE COMMAND

The BYE command terminates operations of BASIC and the system returns to the SCI mode.

FORMAT

BYE

BYE str__exp

where:

str__exp is a valid string expression.

You can follow the BYE command with a string expression. When the BYE command executes, the string expression appears on your terminal. This feature is particularly useful when your BASIC program is running in background mode; the string expression can indicate the status of the program upon completion. In the following example, assume the program is running in background mode:

```
100 ON ERROR 500
.
.
.
490 BYE "NORMAL PROGRAM TERMINATION"
500 A$ = "ERROR DETECTED"
510 BYE A$
```

Editing Capabilities

4.1 INTRODUCTION

TI BASIC provides extensive editing capabilities that allow you to develop programs or modify existing programs. Developing programs entails: writing program lines, adding comments, editing the lines, adding or deleting lines, and copying or moving lines. To make program development easier, BASIC includes numerous editing functions that are implemented through the keyboard. Since the functions and keys are hardware dependent, Appendix I lists the keys and functions by keyboards. Refer to Appendix I to determine which keys on your system perform the functions described in the following paragraphs. Section 4 concludes with a discussion of the OPTION statement, which enables you to mask the function keys.

4.2 WRITING PROGRAM LINES

When the prompting period (.) is displayed, the system is at command level and is ready to receive statements or commands. For program development, you should begin by entering the NEW command. This command clears any other programs from memory, ensuring that the new program will not be combined with an old one in memory. You can enter program statements in either program creation mode or immediate execution mode.

4.2.1 Program Development Mode

In this mode, a line number must precede BASIC statements. You can enter line numbers manually as part of the statement or automatically by using the BASIC command NUMBER. You need not enter line numbers in sequence. The line number identifies the relative position of the statement in the program. Line numbers cannot exceed 32759.

The statement is written to memory when the line terminates. You can save a program developed in this way for future use; to execute it, enter the RUN command. A program developed with line numbers remains in memory until you enter a NEW command, load another program into memory, or terminate BASIC. Therefore, you can move, edit, or delete program lines while developing the program.

EXAMPLE

```
100 PRINT "Hello"
```

4.2.2 Immediate Execution Mode

In this mode, BASIC statements are executed when the line terminates. The statements are not numbered. Since the statements are not written into memory, you cannot save them for future execution; however, you can use the replay function to access the last statement executed.

EXAMPLE

```
PRINT "Hello"
```

Although this execution mode is not practical for most programs, it is useful for debugging. Since the immediate execution method does not disturb memory, a program can occupy memory while its statements are immediately executed. As a result, the status of variables and their relationships can be examined and their values changed before execution continues.

You can write multiple program statements on a single program line. The program statements are separated by two colons. Statements cannot continue from one line to the next.

Program statements are not executed until the statement line terminates. To terminate a line, use one of the following keyboard functions:

- Return
- Display current or preceding line
- Display current or succeeding line
- Return with EOF

4.3 USING COMMENTS

You can use comments within a BASIC program to document the program. The two methods for placing comments in a BASIC program are the remark statement and the tail remark.

4.3.1 Remark Statement (REM)

The REM statement permits you to insert explanatory remarks in a program. The system ignores a REM statement line, allowing you to document the program. While the text that follows REM is ignored, the associated line number can be used in a GOSUB, IF-THEN-ELSE, GOTO, ON-GOTO, ON-GOSUB, ON ERROR, RETURN, RESTORE, RUN, BRKPNT, or UNBRKPNT statement. You cannot use a RUN command to a line number that is a comment in a program saved in LOCK format. Execution continues at the next executable command.

FORMAT

REM text

You can include any of the printable ASCII characters, including the special characters in the text. The following is an example of the REM statement.

EXAMPLE

```
100 REM INSERT DATA IN LINES 900-998. THE FIRST NUMBER IS N,  
110 REM THE NUMBER OF POINTS, THEN THE DATA POINTS THEMSELVES  
.  
.  
.  
200 REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS  
.  
.  
.  
300 RETURN  
.  
.  
.  
520 GOSUB 200  
.  
.  
.  
999 END
```

To insert explanatory remarks following a BASIC statement on the same line, use the multistatement line feature. Separate the statements by a double colon as follows:

```
250 LET Y = 1 :: REM INITIALIZE Y
```

This line includes the remark INITIALIZE Y. The remark does not affect the execution of the program. The REM statement cannot be followed by another statement on the same line; it must be the last statement on the line since anything following REM is treated as part of a remark.

4.3.2 Tail Remark

A tail remark is an exclamation point (!) followed by a remark. A tail remark follows the same conventions as the REM except that a tail remark can follow a statement without being preceded by double colons. The following is an example of a tail remark:

```
250 LET Y = 1 ! INITIALIZE Y
```

4.4 EDITING PROGRAM LINES

You can copy a program into the work file by using the OLD command followed by the pathname of the program. The main part of the program (that is, everything except external BASIC Subprograms) is automatically copied into memory for editing. If you want to bring an external subprogram into memory for editing, enter the EDIT ESUB command (paragraph 3.6.2). To edit the lines of these programs, use the keyboard editing functions. To edit a specific line, use either the display current or preceding line or the display current or succeeding line function. Display the appropriate line, then modify it. The edited line is not changed in memory until the line terminates.

To display a specific line, use the LIST command followed by the line number. If the line number specified in the LIST command does not exist, the prompting period returns. If the line number is unknown, use the LIST command to list all or part of the program.

4.5 ADDING LINES TO A PROGRAM

To add lines to programs that have been saved and loaded into memory by using the OLD command or to programs that are being developed, write program statements with line numbers that do not yet exist in the program. Ensure that each new line number represents the position in the program where that program statement should be placed. Regardless of the order in which the line numbers are entered, the statements are placed in numeric order. If you do not have enough room between line numbers to add the lines desired, resequence the existing statements by using the RESEQUENCE command to make numbers available between lines. The largest number you can specify is 32759.

4.6 DELETING LINES FROM A PROGRAM

To delete a program line from memory, enter its line number and press Return. To delete multiple lines from a program or to delete a file, use the DELETE command.

4.7 COPYING/MOVING LINES IN A PROGRAM

You can copy a program line in memory to another position by changing the line number of the original line. As a result, a copy of the original line is stored in the new line number location. If the original line is then deleted, the result (in effect) is that the original line has been moved. This is an efficient method if only a small number of lines are to be moved; when large blocks of lines are involved, the MERGE command is more efficient. Execute the MERGE command as follows:

1. Access the appropriate program lines in memory, and delete any of those lines that are not to be moved.
2. Resequence the remaining lines. The beginning line number specified should be the line number at which the data will be merged.
3. Save the lines in a temporary file.
4. Call the original program into memory again by using the OLD command.
5. Resequence the lines where the data is being merged to allow room for the new lines.
6. Execute the MERGE command, specifying the temporary file.
7. If the lines are to be moved rather than copied, delete the lines from their original place in the program.
8. Save the file.

4.8 EDITING FUNCTIONS

The editing functions listed below expedite entering and correcting program lines by means of keyboard entries. The functions and the keys that perform them vary according to the operating system and terminal used. Appendix I lists the key functions for the various system configurations.

4.8.1 Space Forward

Use the space forward function to space right on a program line. The characters that are spaced across are deleted.

4.8.2 Move Cursor Right/Move Cursor Left

These two keyboard functions move the cursor to the left or right. When the cursor moves across a character, the character remains unchanged.

4.8.3 Back Tab

This function moves the cursor to the left eight character positions each time you press the function key. Characters that are tabbed over remain unchanged.

4.8.4 Return

This function terminates a program line. The program line is either executed immediately (if in immediate execution mode) or stored in memory (if in program creation mode).

4.8.5 Display Current or Preceding Line

This function displays the specified program line. If the specified line does not exist, the next lower-numbered (preceding) program line appears. This function requires that the program lines already exist in memory (whether previously created or brought into memory by using the OLD command). Using the function again writes the current line, including any corrections, into memory and displays the preceding line. The following example illustrates the use of this function. Assume that the following program lines exist in memory.

```
10 PRINT "HERRO"  
20 PRINT "GOODBYE"  
30 END
```

If, after listing the program above, you want to make a correction to line 10, you enter the number 10 following the prompting period and press the function key. Line 10 appears. You can then replace "HERRO" with "HELLO". When you press the Return key, the edited line is written into memory, replacing the former line. If you enter 5 and press the function key, the prompting period followed by the number 5 remains on the screen since no program statement numbered 5 exists in memory and no lower-numbered program statement exists. You can then add a program statement as line 5, if desired. If you enter line number 40 when using this function, line number 30 appears on the screen since no line 40 exists in memory and line 30 is the next lower-numbered line.

Entering any number higher than the highest-numbered line in the program displays the last line of the program. If this function is used while a program line appears, the program line, including any corrections made to it, is written into memory, replacing the appropriate line in memory. The next lower-numbered program line is then displayed.

4.8.6 Display Current or Succeeding Line

This function works in the same way as the display current or preceding function with one exception: if the specified line number does not exist, the next higher-numbered line is displayed. Entering line number 1 and using this function results in the display of the lowest-numbered line in memory. If the number specified does not exist and no higher-numbered line exists, the prompt character followed by the line number specified remains on the screen. You can enter a program statement at that line number, if desired.

4.8.7 Insert Character

The insert character function inserts characters in a program line. Every character entered after you press the function key is entered to the left of the cursor. For each character entered, the cursor, the character at the cursor position, and the characters to the right of the cursor are shifted one character position to the right. The number of characters that can be inserted is limited by the line length. When the line is full, no more characters can be entered. The system will not shift characters past the end of the line during an insert. To terminate the insert character mode, use any other keyboard function except repeat and space forward.

4.8.8 Delete Character

The delete character function deletes the character at the cursor position. All characters to the right of the deleted character are shifted one character position to the left.

4.8.9 Erase Input

The erase input function erases all characters on the current line; on printer terminals, the printhead is positioned at the beginning of the line.

4.8.10 Erase Field

This function erases the current line from the cursor position to the end of the line.

4.8.11 Tab

The tab function advances the cursor to the end of the displayed program line. You can then enter data or additional statements on the program line.

4.8.12 Repeat

The repeat function is used in conjunction with the other keys on the keyboard. If you press and hold the function key and then press the key whose action is to be repeated, the specified action repeats until you release the function key. On some keyboards, the repeat function is not a separate function key; these keyboards have *typamatic* keys that repeat the character if you hold the key down for longer than one second.

4.8.13 Replay

The replay function redisplay the last line entered from the keyboard while at command level. For example, if you entered the RUN command and the program executed, using the replay function at the end of the program redisplay RUN. You can then reexecute the program using the return function. The replay function operates at command level only.

4.8.14 Calculate

The calculate function evaluates an expression immediately. You can evaluate any legal expression by entering the expression in response to the prompting period and pressing the function key. The result appears following an equal sign. The following is an example:

```
.123*2/10 = 24.6
```

The calculate function operates at command level only.

4.8.15 Break Execution

This function stops execution of a program immediately. When you use this function, the following message appears and the prompting period reappears:

```
STOPPED #1 IN line__num
```

This function enables you to examine variable values. If you modify the program, you must restart it with a RUN command. When you use this function at command level, it functions like the erase input function.

4.8.16 Resume Execution

This function resumes execution of a program after the break execution function has been used.

4.8.17 Step

Use the step function after a breakpoint has caused a break in execution or when you use the break execution function while a program is executing. The step function resumes execution of one program statement at a time. Usually, this function is used for debugging programs.

4.8.18 Reset Cursor

The reset cursor function repositions the cursor at the beginning of a displayed program line, command, or input.

4.8.19 Suspend Execution on Output

The suspend execution on output function suspends execution of the BASIC system the next time it attempts to display any data on the terminal. This feature is most useful during a LIST of a large program. Pressing the function key or any other key a second time causes the system to resume execution.

4.8.20 Return with EOF

This function operates like the return function in that it terminates a program statement; however, the return with EOF also sets the EOF flag for the unit number associated with the screen. You can test the condition of the flag using the EOF function. The next input/output (I/O) to or from the screen resets the flag.

4.9 OPTION STATEMENT

The OPTION statement has two forms: OPTION 1 and OPTION 2. OPTION 1 allows you to disable the function keys so that they cannot affect a program while it is executing. OPTION 2 allows you to see which keys have been masked.

The form of the OPTION statement is either of the following:

FORMATS

OPTION num__exp__1,num__exp

where:

num__exp__1 is a numeric expression that equals 1.

num__exp is an expression that indicates the keys to be inhibited.

or

OPTION num__exp__2, num__var__name

where:

num__exp__2 is a numeric expression that equals 2.

num__var__name is a numeric variable name that returns the value of the current mask.

4.9.1 OPTION 1

Using OPTION 1, you can inhibit one or more of the function keys; you can then use these keys as data keys. Table 4-1 shows the bits of the numeric expression (num__exp) associated with the keys.

Table 4-1. Bit Positions and Values Associated with Keys

Bit Position	Function Key	Numeric Value	Function
14	F1	2	Calculate
13	F2	4	Replay
12	F3	8	
11	F4	16	
10	F5	32	
9	F6	64	
8	F7	128	Resume Execution
7	F8	256	Step
6	Command	512	Command
4	Attention/ CTRL X	2048	Break Execution
0-3,15,5	Unused	Unused	Unused

The numeric expression is used as a mask to indicate whether the associated key is a data key or a function key. When no numeric value is given, the default value of 2048 is used, causing the break function to be masked.

For example, to inhibit the step and replay functions, bits 7 (F8) and 13 (F2) must be set. Bit 7 (F8) equals 256. Bit 13 (F2) equals 4. Therefore, each of the following OPTION statements inhibits the step and replay functions.

```
100 OPTION 1,256 + 4
100 OPTION 1,260
```

To reset the inhibited functions, execute the OPTION statement, specifying a value that resets the bits previously set or specifying a value of zero (which resets all of the bits). The following statement resets all bits:

```
OPTION 1,0
```

Note that the bits set by the OPTION statement are not reset using the edit functions or the commands. The mask established by the OPTION statement is maintained only during execution of the program. When the program terminates or runs another program, the mask is cleared.

4.9.2 OPTION 2

Using OPTION 2, you can specify a numeric variable name to receive a copy of the contents of the current mask; the current mask contains a numeric value representing the sum of the bit values of the masked keys. You can check this numeric variable name to determine which keys have been inhibited. The following example illustrates the use of the OPTION 2 statement:

```
100 OPTION 2,MSK  
110 PRINT MSK
```


Data and Expressions

5.1 INTRODUCTION

BASIC program statements contain data and expressions. BASIC handles two types of data: numeric and character string. The value of this data can be either constant or variable within a program. An expression is a single item of data or two or more data items joined by operators such as + or - to combine constants and variables. After learning how TI BASIC handles data and expressions, you will learn how BASIC evaluates expressions.

5.2 DATA

BASIC handles two types of data: numeric and character string. Both types can occur as constants or variables. Numeric data can be stored in an integer, floating-point, or decimal format. String data stores nonnumeric information such as words or other sequences of characters. String data consists of alphabetic, numeric, and special characters.

5.3 CONSTANTS

A constant is a value that cannot change; it retains its intrinsic value throughout program execution. Constants can be either numeric or string. The following are examples:

EXAMPLES

```
10  
3.1415926  
"This is a string constant."
```

The first two examples are numeric constants, while the third is a string constant.

5.3.1 Numeric Constants

A numeric constant can be a positive or negative number; optionally, it can contain a decimal point and/or an exponent. An exponent, symbolized by the letter E, signifies that the number preceding the E is to be multiplied by the integer power of 10 indicated by the number following the E. For example, 1.50E2 means 1.50 times 100, or 150. Constants should have a value no larger than 9.999999999999999E + 127 and no smaller than 1E - 128. This condition holds for both initial input and internal data. The following examples are valid numeric constants in a BASIC program:

EXAMPLES

```
1.0
120.0
0.234
1.34967
12345678.0
12345678.91234
-2.0
-5.55E5
8.02
6.00
-.543298721
-123.4567891234
```

The following are not valid numeric constants in a BASIC program:

EXAMPLES

```
10X2
2.22.
5,280.66
```

The first invalid example contains an illegal character (X is not numeric), the second contains two decimal points, and the third contains an illegal comma.

5.3.2 String Constants

String constants are sequences of printable ASCII characters defined in the BASIC character set (see Appendix B) and enclosed in quotes. A string constant is an explicit representation of data and remains unaltered during execution of the program. Although a numeric character can appear in a string, an attempt to perform arithmetic on a string results in an error. Likewise, an attempt to perform a string operation on a numeric value results in an error. A quote can appear within a string if it is entered as two contiguous quotation marks (""). The following are valid string constants.

EXAMPLES

```
"CONSTANT"
"123.45"
";;"
"HE SAID, ""HI. """
```

The following are not valid string constants:

EXAMPLES

```
'NOT A STRING'
"BASIC
"HE SAID, "HI.""
```

The first invalid example contains apostrophes rather than quotation marks; the next example lacks the second quotation mark; the third needs two contiguous quotation marks on each side of the dialogue (it should be "HE SAID, ""HI. """).

5.4 VARIABLES

Variables are symbols whose values are defined during the execution of a BASIC program. For example, in a program that processes a payroll, an individual's pay rate can be constant over a period of time; however, the number of hours worked in any given week would be a variable. Variables can be numeric or string.

Variables can be grouped and accessed by one name. The grouping is referred to as an *array*, and the name by which the variables are accessed is called the *array name*. A particular variable is accessed in the array by specifying the array name followed by a subscript. Arrays are described later in this section.

5.4.3.1 The REAL Statement. The REAL statement assigns the real data type to a variable or list of variables. Real variables must be less than $1E + 128$; they can include up to 14 digits. You should use a real variable when the evaluation of an expression can result in a value greater than 32,767 or less than -32,768 or in a fractional value.

FORMATS

REAL ALL

REAL var__1, var__2, . . .

where:

var__1 and var__2 are numeric variable names or array declarations (described later).

The following example declares the variables A, B, C, and D to be of the real data type. REAL is the default specification.

```
10 REAL A, B, C, D
```

5.4.3.2 DECIMAL Statement. The DECIMAL statement assigns the decimal data type to a variable or list of variables. Variables of this type are identical to real variables in internal representation except that you can force rounding of the values on output. (Values less than or equal to four round down; values greater than or equal to five round up.) This type of variable is useful for computations where limited accuracy is required (for example, in developing financial reports in which fractions of a cent are not to be retained).

FORMATS

DECIMAL (precision) ALL

DECIMAL (precision) var__1, var__2, . . .

where:

var__1 and var__2 are any numeric variable names or array declarations.

precision is an optional value specifying the number of digits of accuracy to maintain (the range for precision is - 15 to 15).

The precision value specifies the number of digits to be retained to the right of the decimal point. For example, if the precision value is 2, all digits beyond the hundredths place are rounded off. If the precision value is -3, values are rounded off to the nearest thousand. If the precision value is 0, only the number of units is retained.

If the precision value is omitted, the variables default to integer values on output. The precision specification can be positive or negative.

In the following example, the variables A, B, C, and D are declared to be of the decimal data type with only the first two places to the right of decimal point retained on output.

```
10 DECIMAL (2) A, B, C, D
```

The following statement declares that all numeric variables not appearing in another type statement are to be of decimal data type with two-digit accuracy maintained to the left of the decimal point for output.

```
10 DECIMAL (-2) ALL
```

For example, the number 5343 would be displayed as 5300.

5.4.3.3 INTEGER Statement. The INTEGER statement assigns the integer data type to a variable or list of variables. The range of values for an integer variable is from -32,768 through + 32,767. Integer values cannot contain fractions.

FORMATS

```
INTEGER ALL
```

```
INTEGER var__1, var__2
```

where:

var__1 and var__2 are any numeric variable names or array declarations (described later).

In the following example, the variables A, B, C, and D are declared to be of the integer data type.

```
10 INTEGER A, B, C, D
```

5.4.4 String Variables

String variables are string values that can be altered during program execution. String values are sequences of the BASIC character set. A string can contain a maximum of 255 characters. Usually, strings contain nonnumeric information such as names and part descriptions. They are also used to print messages and describe numeric quantities.

5.4.5 String Variable Names

String variable names follow the same rules that apply to numeric variable names except that they must always end with a dollar sign (\$). The following are examples of string variable names:

EXAMPLES

```
A$
NAME$
Z123$
ADDRESS$
```

5.5 VALUE ASSIGNMENT

The LET statement assigns values to both numeric and string variables. It can be used to assign the initial value of a given variable or to change the value of a variable during execution of the program.

5.5.1 Arithmetic LET

The arithmetic LET statement assigns a value to a variable or changes the value of a numeric variable during program execution.

FORMATS

```
LET num__var = num__exp
```

```
num__var = num__exp
```

where:

num__var is a numeric variable.

num__exp is a numeric expression.

The value of the expression on the right side of the equal sign is assigned to the variable on the left of the equal sign. The variable can be either subscripted or unsubscripted. The variable to the left of the equal sign retains the assigned value until another BASIC statement redefines it. The word LET is optional.

In the following example, the first statement assigns the value of 5 to XY, the second increments A by 1, the third assigns 69 to XZ, and the fourth assigns XZ to Z9:

EXAMPLE

```
100 LET XY = (2*3) + (4 - 5)
110 LET A = A + 1
120 XZ = 69
130 Z9 = XZ
```

5.5.2 String LET

The LET statement can also assign values to subscripted or unsubscripted string variables.

FORMATS

```
LET str__var = str__exp
```

```
str__var = str__exp
```

where:

str__var is a string variable name as previously defined.

str__exp is any valid string expression.

The keyword LET is optional. The following is an example of a string LET:

EXAMPLE

```
100 LET B$ = "STRING ONE"  
110 LET A$ = B$
```

5.6 ARRAYS

Often it is advantageous to group variables of the same type together under one name. In BASIC this grouping is known as an *array*, which is accessed by an *array name*.

To access an individual element of an array, specify its position in the array with a subscript enclosed in parentheses after the array name. For example, to reference the element in the third row and second column of an array named SURVEY, use the term SURVEY(3,2).

5.6.1 DIM Statement

Use the DIM statement to declare arrays. The DIM statement defines the number of data elements that can be contained in the specified array.

FORMATS

DIM array__name (integer)

DIM array__name (integer, integer, . . .)

where:

array__name is any variable name.

integer(s) is the unsigned integer constant specifying the dimensions of the array.

The following is an example of a two-dimensional array with space reserved for 121 elements.

EXAMPLE

```
DIM ARRAY1(10,10)
```

The integer specifies the maximum value a subscript can have in that position. The number of integers specified is the number of dimensions of the array. Normally, the subscript value ranges from zero up to the value of the integer. Thus, a one-dimensional array has one more element than the value of the integer. The total number of elements in an array equals the product of the number of elements along each dimension. Thus, an array declared as F(3,4,5) has $(3 + 1) * (4 + 1) * (5 + 1)$ or $4 * 5 * 6$ or 120 elements.

More than one array name can appear in a DIM statement, provided that each array name is separated from the previous array by a comma.

EXAMPLE

```
DIM A(5,5),B(20),C(9,2,2,9)
```

In the following example, statement 10 defines an array named HOURS with elements numbered 0 through 10. To access the data element stored in the third element, you should reference HOURS(2).

EXAMPLE

```
10 DIM HOURS(10)
```

The following example defines an array with nine possible locations: (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2).

EXAMPLE

```
10 DIM SQUARE(2,2)
```

Similar procedures are used for arrays with more than two dimensions. BASIC automatically creates arrays with a lower boundary of zero and an upper boundary of 10 for each dimension. Therefore, one-dimensional arrays that contain 11 or fewer data elements need not appear in a DIM statement. When all dimensions of a multidimensional array have an upper boundary of 10 or less, the array need not appear in a DIM statement. In all other cases, the array must be specifically defined in a DIM statement or in a type declaration that declares the size of the array. The following are examples:

EXAMPLES

```
10 DIM A(20,20)
20 REAL B(5,2,7)
```

NOTE

Although you need not use the DIM statement for arrays with fewer than 10 elements for each dimension, doing so can save considerable memory space. For example, an array of real numbers defined as (4,4,4,4) reserves 5000 bytes of memory ($5*5*5*5*8$). If you accept the default, the system attempts to reserve 117,128 bytes of memory ($11*11*11*11*8$).

You can also dimension arrays with REAL, DECIMAL, and INTEGER statements. Such arrays have the same characteristics as variables declared with these types of statements. Arrays can also be dimensioned for strings. String arrays follow the same rules for definition and access as numeric arrays. In the following example, A\$ is defined as an array containing 20 strings, that is, A\$(0), A\$(1), A\$(2), . . . A\$(19).

```
100 DIM A$(19)
```

Each string element of A\$ follows the same rules that govern a string.

NOTE

A DIM statement must precede all references to the array it defines.

The following examples illustrate correct and incorrect use of the DIM statement and a variable in the dimension array.

EXAMPLE

CORRECT:

```
100 DIM A(100)
110 LET A(50) = 9
```

INCORRECT:

```
100 LET A(50) = 9
110 DIM A(100)
```

5.6.2 OPTION BASE Statement

When an array is named in a program, whether dimensioned in a DIM statement or otherwise, the element zero, which is A(0) in a one-dimensional array, is created by BASIC unless the OPTION BASE statement is used.

FORMAT

OPTION BASE n

where:

n is 1 or 0.

The value of n declares the minimum value of all array subscripts. If no OPTION BASE statement occurs in a program, the default is zero. If an OPTION BASE statement specifies that the lower bound for all array subscripts is one, no DIM statement can specify a lower bound of zero. A program can contain only one OPTION BASE statement. If used, the OPTION BASE statement must occur in a lower-numbered line than any declarative or executable statements that reference array elements. The type ALL statement and the MERGE statement are exceptions to this rule.

5.7 VIRTUAL ARRAYS

BASIC supports array file structures, allowing data stored in files to be accessed as if it were stored in memory. A virtual array is referenced within a program in the same manner as a memory-resident array. However, a reference to a virtual array forces the system to calculate the disk address of the element and read that element into memory, if it is not already present. Also, modification of any virtual array element forces the system to write the updated element back to the appropriate disk address.

Virtual array files are relative record files (Section 6). Generally, you should use virtual arrays for large or infrequently accessed data. Virtual arrays can contain as many as 65,000 elements. The lower bound (the minimum value of all array subscripts) is zero unless changed to one by using the OPTION BASE statement. The original definition of the file determines the upper bound.

You can use the ASSIGN statement to open a virtual array and a special form of the CLOSE statement to close the virtual array.

5.7.1 ASSIGN Statement

The ASSIGN statement establishes program access to virtual arrays or creates a virtual array file if one does not exist. The ASSIGN statement assigns a disk file to a virtual array. It also dimensions the array. Note that the USING clause is always required with this statement.

FORMAT

```
ASSIGN pathname USING array__name (integer, . . .) max__size
```

where:

pathname	is any legal relative record file pathname or string variable containing the pathname. If the file does not exist, it is created.
array__name	is the name to be given to the array.
integer(s)	specifies the number and size of the dimensions, as with the DIM statement.
max__size	is optional and specifies the maximum size of a string for a string array variable. It consists of an asterisk (*) followed by a number.

An example of the ASSIGN statement is as follows:

EXAMPLE

```
ASSIGN "DS01.MYFILE" USING PCNT(100,10)
```

DS01.MYFILE is the name of the file on which the virtual array is located. It must be specified by any legal string expression. The keyword USING precedes the declaration of the array as it would appear in the DIM statement. The array name can be preceded by a type declaration, for example, USING INTEGER JULIANDATE (365,4).

An ASSIGN statement must precede any reference to the virtual array, as shown in the following example.

EXAMPLE

```

100 ASSIGN "DS01.MYFILE1" USING NAME$(500)
105 ASSIGN "DS01.MYFILE2" USING INTEGER ENUM(500)
110 FOR I = 1 TO 3
120 PRINT NAME$(I),ENUM(I)
130 NEXT I
140 END

JOHN DOE          345-6768
JANE DOE          876-4365
MARY DOE          677-4344

```

This example prints names that have been previously stored in the virtual array assigned in line 100, along with employee numbers stored in the virtual array assigned in line 105. You can also specify the number of characters or the length of a string in the ASSIGN statement.

Virtual arrays that contain string data have a maximum string size. Elements of the virtual array cannot be larger than this maximum. In the following example, each string in array NAME\$ can contain a maximum of 30 characters.

EXAMPLE

```

100 ASSIGN "DS01.MYFILE" USING NAME$(500)*30

```

If no length is specified, string virtual arrays default to a maximum of 18 characters. The type of virtual array is the default data type if either of the following is true:

- The type of virtual array is not explicitly defined by a type specification immediately preceding the numeric declaration.
- The virtual array name does not end with a dollar sign, indicating a string virtual array.

If the array file already exists, the type specification must be the same.

5.7.2 CLOSE Statement

To close a virtual array file, specify the array name in a CLOSE statement.

FORMAT

```
CLOSE array__name
```

where:

array__name is the name assigned to the virtual array.

The following statement closes the file associated with the virtual array NAME\$.

EXAMPLE

```
CLOSE NAME$
```

NAME\$ is the array name associated with the file in the ASSIGN statement of the preceding example. To reopen a virtual array file, execute an ASSIGN statement with the same array or different array name. A virtual array file implicitly closes when a STOP, END, or RUN command executes.

5.8 EXPRESSIONS

Expressions are formed by combining constants and variables with symbols, called operators. Like variables and constants, expressions are evaluated to either numeric or string values.

If an expression contains any variables, its value depends on the values of the variables at the time the expression is evaluated. For example, $A + 10$ is an arithmetic expression consisting of the variable A , the arithmetic operator $+$, and the numeric constant 10. If A equals 5 when the expression is evaluated, the value of the expression is 15.

5.9 OPERATORS

Operators are special symbols within the TI 990 BASIC character set and are used to combine variables and constants. An operator requires one or two operands. Each operator is categorized as arithmetic, logical, string, or relational, depending on the function it represents.

5.9.1 Arithmetic Operators

Combinations of variables, constants, arithmetic operators, and relational operators are used to perform arithmetic. The arithmetic operators (Table 5-1) specify addition, subtraction, multiplication, division, negation, and exponentiation.

Table 5-1. Arithmetic Operators

Operator	Meaning	Example
-	Negation	-B
+	Addition	A + B
-	Subtraction	A-B
*	Multiplication	A*B
/	Division	A/B
^	Exponentiation	X^2

5.9.2 String Operator

Use the string operator (&) to concatenate. A string expression consists of one or more string variables, constants, or function references separated by the string operator, as shown in the following example. In this example, the string operator assigns the value ABCDEFGHI to A\$.

EXAMPLE

```
80 B$ = "DEF"
90 A$ = "ABC" & B$ & "GHI"
```

5.9.3 Relational Operators

BASIC provides six relational operators for comparing two expressions, which can be numeric or string. You can compare string expressions only to string expressions and numeric expressions only to numeric expressions. The relational operators (Table 5-2) are all of the same precedence. A relational expression equals -1 if true and 0 if false. You can assign the value of a relational expression to a numeric variable. You can also use relational operators in arithmetic expressions.

Table 5-2. Relational Operators

Operator	Meaning	Example
=	Is equal to	A = B
<	Is less than	A < B
>	Is greater than	A > B
< = or = <	Is less than or equal to	A < = B
> = or = >	Is greater than or equal to	A > = B
< > or > <	Is not equal to	A < > B

Two strings are equal only if they have the same length and contain identical sequences of characters. For example, in the following, C\$ is equal to D\$ because each contains the same number and sequence of characters.

EXAMPLE

```

10  A$ = "AAA"
20  B$ = "BBB"
30  C$ = "AAABBB"
40  D$ = A$ & B$

```

When strings are compared, the characters within the strings are compared on the basis of their ASCII codes (Appendix B). Thus, "\$ZOO" is less than (<) "ANT" because the dollar sign has an ASCII code less than that of A; similarly, "car" is greater than (>) "CAR" because the ASCII code for a lowercase c is higher than that of an uppercase C. If the strings are of unequal length and differ only because of additional characters, the longer string is considered greater than the shorter string. For example, "SUNNY" is greater than "SUN".

When numeric or string values are compared using the relational operators, the result is 0 when the relationship is false and -1 when it is true. The following example provides several relational expressions and the results of their evaluations.

EXAMPLE

```
PRINT 5*7>6 ; 22<=2.2/1 ; "ABC"&"DEF"="ABCDEF" ; .5>3.14
```

```
-1 -1 -1 0
```

5.9.4 Logical Operators

The logical operators are NOT, AND, and OR. The result of a logical operation is a logic value, which indicates that the result of the operation is either true or false. BASIC represents logic values as zero (false) and nonzero (true). To avoid confusion with arithmetic operations, logic values in Table 5-3 are indicated by T (for true) or F (for false).

Table 5-3. Truth Table

Value of X	Value of Y	Value of NOT X	Value of X AND Y	Value of X OR Y
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

NOTE

The logical operators are also available for bit manipulation of integer (16-bit) results. Floating-point values are converted to integer form before being used with these operators. Appendix F contains additional information on how to use the logical operators.

5.9.4.1 Logical NOT. The NOT operator yields a result opposite to the logic value of the expression. Thus, if an expression is evaluated as true, the NOT operator yields a value of false and vice versa.

5.9.4.2 Logical OR. The logical OR operator produces a logic value of true if either of the expressions is evaluated as true.

5.9.4.3 Logical AND. If the values of X and Y are both true, the AND operator yields a logic value of true. If either or both of the values are false, the AND operator produces a logic value of false.

5.9.4.4 Logical Operator Example. The order of precedence of the logical operators is NOT, AND, and OR, as shown in the example below. You can use parentheses to override this order. Although the logical operators have differing precedence, the relational operators all have the same precedence.

EXAMPLE

```

100 A = 4 :: B = 6 :: C = 2
110 IF NOT( A > 3) THEN PRINT "TRUE" ELSE PRINT "FALSE"
120 IF (A < B) AND (B < C) THEN PRINT "TRUE" ELSE PRINT "FALSE"
130 IF (A < B) OR (B < C) THEN PRINT "TRUE" ELSE PRINT "FALSE"
140 IF NOT (A < B) AND (B < C) THEN PRINT "TRUE" ELSE PRINT "FALSE"
150 IF NOT (A < B AND B < C) THEN PRINT "TRUE" ELSE PRINT "FALSE"

FALSE
FALSE
TRUE
FALSE
TRUE
    
```

5.10 PRECEDENCE OF OPERATOR EVALUATION

To simplify the discussion of the precedence of operations, the operators are assigned the priority numbers given in Table 5-4. One (1) has the highest priority.

Table 5-4. Numeric/Logical Expression Priority

Arithmetic Operation	Priority	Symbol
Exponentiation	1	^
Negation (unary minus)	2	- (e.g., -X)
Multiplication	3	*
Division	3	/
Addition	4	+
Subtraction	4	-
Relational operators	5	=, <, >, <=, >=, <>
Logical NOT	6	NOT
Logical AND	7	AND
Logical OR	8	OR

Expressions are evaluated from left to right, using the priorities in Table 5-4, unless a portion of the expression is in parentheses. When a portion is enclosed in parentheses, that portion is treated as a single element and evaluated before being associated with the remainder of the expression. Evaluation within parentheses is made with the same left-to-right priority. Expressions can also contain nested parentheses (a pair of parentheses inside a pair of parentheses). The expression contained within the innermost parentheses is evaluated first.

5.11 EVALUATION OF EXPRESSIONS

All expressions are evaluated either to a numeric or string value. Note that if the expression evaluates to a logic value, that value is stored as a numeric value; this numeric value represents the logic value (true or false).

The order of evaluation of any expression is based on the priority of the operators in the expression. The operator evaluated last determines the expression type. Thus, if the last operator evaluated is an arithmetic operator, the expression is an arithmetic expression that produces a numeric result.

5.11.1 Arithmetic Expressions

Subject to two conditions, any combination of numeric variables (subscripted or unsubscripted); numeric constants; numeric functions; and arithmetic, logical, and relational operators constitutes a valid arithmetic expression. The two conditions are as follows:

- An arithmetic operator must be the last operator evaluated in the expression.
- No two variables, constants, or operators can appear in succession. The only exception is that the unary plus or unary minus can appear following another operator (for example, X^*-Y).

The following expressions are valid arithmetic expressions.

EXAMPLES

B
 $A + 7.05/A$
 $X*Y/2$
 $-B/2.3$
 $X + Y + Z$
 $(A \text{ AND } B) + 1$
 $(A = B)^*-1$

Note that a single numeric variable or numeric constant is a valid expression. The following are not valid arithmetic expressions:

EXAMPLES

```
7.05A + B*2
123.45X
/
10.2Y
ABC + /XYZ
```

Mathematical and string functions can also appear in arithmetic expressions. The following program segment includes several examples of the use and evaluation of expressions:

EXAMPLE

```
100 A = 2
110 B = 3
120 C = 4
130 D = B + C/2
140 E = B^2 + C/A^2
150 F = C^2 - C/A
160 G = A*(A + B^2) - 22
170 H = A^^^B
```

Everything to the right of the equal sign is an expression.

At line 130, C/2 equals 2 and is added to B to obtain 5.

At line 140, B raised to the second power is 9, A raised to the second power is 4, C divided by this intermediate result is 1, and the result added to 9 equals 10.

At line 150, C raised to the second power is 16, C divided by A is 2, and 2 subtracted from 16 equals 14.

At line 160, B raised to the second power is 9, A added to this intermediate result is 11, 11 multiplied by 2 is 22, and 22 subtracted from 22 equals 0.

At line 170, A raised to the second power is 4, and 4 raised to the third power is 64. Note that exponentiation is performed from left to right.

The following example illustrates the use of nested parentheses. In this example, C divided by A is 2, 2 multiplied by A is 4, and 4 added to the product of B and C is 16. This intermediate result is then multiplied by C, resulting in 64, and 64 is added to A, resulting in 66.

EXAMPLE

```

100 A = 2
110 B = 3
120 C = 4
130 D = A + C*(A*(C/A) + B*C)

```

NOTE

BASIC performs all arithmetic in real mode regardless of the types of variables involved. Therefore, all intermediate results in an expression are real unless explicitly typed using intrinsic functions.

In the following example, the subexpression A/B in statement 30 produces a real result even though A and B are integer variables. This intermediate result is added to 100 and then rounded to an integer value before being assigned to variable E.

EXAMPLE

```

10 INTEGER A,B,E
20 INTEGER D,C,F
30 E = 100 + A/B
40 F = C*(C/D)

```

For a result other than a real intermediate result, use the intrinsic function INT. To maintain the same number of digits of accuracy throughout the calculation, the expression at statement 40 should be divided into the following two statements:

```

40 F = C/D
50 F = F*C

```

In this way, no real intermediate value enters into the calculation.

5.11.2 Logical Expressions

BASIC evaluates a logical expression to determine whether the relationship is true or false. The expression $A > B \text{ AND } C < D$ is evaluated as true if the value of A is greater than the value of B and the value of C is less than the value of D. BASIC first evaluates the two relational expressions in accordance with priority to determine their logical value. Then, the logical AND is applied to the values of the two subexpressions, yielding a result for the entire expression.

The expression $A = B \text{ OR } C < = D$ is evaluated as true if either of the relational expressions holds true. Therefore, the logical expression is said to be true if the value of A is equal to the value of B, or the value of C is less than or equal to the value of D. If both expressions are evaluated as false, the entire logical expression is evaluated as false. However, if only one of the expressions is true, the entire expression is evaluated as true.

You can use the relational negator NOT to complement the value of a relational expression. Therefore, $\text{NOT } A > B$ has the same result as $A < = B$. The NOT operator cannot immediately precede another logical operator; therefore, the expression $A \text{ NOT } > B$ is illegal.

5.11.3 String Expressions

You can use string constants and variables with the string concatenation operator (the ampersand) and the intrinsic string functions (Section 8) to produce string expressions. String expressions result in string values; consequently, you cannot use string expressions when numeric or logical values are required. In the following example, A\$ is combined with B\$ by using the ampersand.

EXAMPLE

```
100 A$ = "12345"  
110 B$ = "67890"  
120 C$ = A$ & B$  
130 PRINT C$
```

RUN

1234567890

5.11.4 Relational Expressions

You can use the relational operators with constants, variables, and arithmetic and logical operators to form relational expressions. However, the relational operator must be the last operator evaluated in the expression. The evaluation of a relational expression yields a logic value of either true or false. Although relational expressions are frequently used with the IF . . . THEN . . . construct, they can be used anywhere an expression can be used.

Input/Output (I/O) Statements

6.1 INTRODUCTION

I/O statements allow the program to communicate with peripheral devices and data files. These statements permit you to enter data into the program and enable the program to output data to you. Generally, the information input or output by a statement is referred to as a *record*, and a collection of records is a *file* (analogous to records kept in a conventional file drawer). In the BASIC system, files are stored on disks that can be both read and written to by a BASIC program.

You can store multiple files on a single disk.

After reviewing the three types of files that BASIC supports, this section describes the BASIC statements used to manipulate I/O. Files are made available to a BASIC program by means of the OPEN statement. To access records, use the PRINT, DISPLAY, INPUT, and ACCEPT statements. Use the RESTORE statement to position the file position pointer to a particular record. Use the SCRATCH statement to delete records and the REPRINT statement to modify records within a key indexed file (KIF). The CLOSE statement ensures that all data written to the file is recorded on the disk.

6.2 FILE ORGANIZATION

Every file has a file organization attribute that specifies the logical structure of the file. BASIC supports sequential, relative record, and key indexed file organization, as indicated by the keywords SEQUENTIAL, RELATIVE, and KEYED. Refer to the manuals that accompany your operating system for instructions on file creation.

The OPEN statement makes a specific file or device available to a BASIC program. OPEN can also create a file (except for KIFs) for program use. When the OPEN statement does not specify a file organization, sequential organization is assumed. Devices handle records sequentially; thus, their organization is sequential by default.

6.2.1 Sequential

In sequential files, records are read and written in sequential order, beginning with the first record in the file. Additional records can be appended to the end of an existing file.

Peripheral devices attached to a system are treated as sequential files. These devices can be opened, and records can be read from and written to the device (if the device permits) in sequential fashion.

6.2.2 Relative Record

Relative record files permit both sequential access and random record access. To access a record nonsequentially, you must specify its numeric position in the file, relative to the beginning. The first record in a file is record number 0. Thus, to access the tenth record in a relative file, you would specify record number 9. The paragraph entitled "Opening a Relative Record File" provides an example.

6.2.3 KIF

Although the records of sequential and relative record files are read by identifying their position in the file, the records of KIFs are read by identifying a portion or field of the record. Thus, instead of searching sequentially through a KIF for the desired record, you need only identify the field you want. These fields are known as *keys*, and their contents are *key values*. A key is defined at the file level and, therefore, applies to every record in the file. The first key defined is the primary key; other keys are secondary keys. In a personnel file, for example, the primary key might be the employee identification number, while secondary keys might be last name, first name, telephone extension, home address, and so on.

BASIC cannot create KIFs; you must create them in the SCI mode by using the Create Key Indexed File (CFKEY) command.

6.3 OPEN STATEMENT

Before a BASIC program can access a file, the file must be opened. Opening a file associates a number, the *unit number*, with a specified file. Records in the file can be accessed until the file is closed. A unit number assigned to one file cannot be assigned to another file until the first one is closed. All opened files are closed upon execution of a NEW, OLD, or RUN command.

The OPEN statement makes a specific file or device available to a BASIC program. OPEN can also create a file (except for KIFs) for program use. OPEN identifies the file to be referenced by specifying which volume or disk the file resides on and the file name. It also indicates the unit number assigned to the file or device while it remains open. The OPEN statement can also create an output file to receive data supplied by the BASIC program.

FORMAT

OPEN # unit__number: pathname, attribute__list

where:

unit__number is any numeric value or expression that has a value greater than 0 and less than 256.

pathname is the file pathname (or device name) to be opened or created. It can be expressed as a pathname or device name within quotations, such as "DS01.MYFILE", or as a string variable, such as NAME\$ (where NAME\$ = "DS01.MYFILE").

attribute__list can contain any of the following items (in any order):

File organization attribute
 File/device format attribute
 File/device record length attribute
 File life
 File access attribute

A comma is required between each attribute in the list.

The following paragraphs describe the file attributes you can include in the OPEN Statement.

6.3.1 OPEN Statement with File Organization Attribute

If the OPEN statement does not specify a type of file organization, sequential is the default if the file does not already exist. If you attempt to open a relative record file or KIF without specifying the file organization attribute, an error message appears.

6.3.1.1 Opening a Sequential File or Device. The following statements are equivalent. They open a sequential file named DS01.MYFILE on unit number 1.

EXAMPLE

OPEN #1: "DS01.MYFILE",SEQUENTIAL, OUTPUT

OPEN #1: "DS01.MYFILE", OUTPUT

Devices can be accessed by BASIC as in the following example.

EXAMPLE

```
OPEN #7:"LP01"
```

The line printer is opened as unit number 7. Since data cannot be read from a line printer, only output can be performed to that device.

6.3.1.2 Opening a Relative File. When opening a relative record file, you must specify the attributes RELATIVE, INTERNAL and FIXED. The following example opens a relative record file named DS01.MYFILE on unit number 1.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",RELATIVE,INTERNAL,FIXED,OUTPUT
```

6.3.1.3 Opening a KIF. BASIC cannot create KIFs; you must create them in the SCI mode by using the Create Key Indexed File (CFKEY) command. After using SCI to create a KIF named DS01.KIF, you can open it in BASIC as follows:

EXAMPLE

```
OPEN #5:"DS01.KIF", KEYED
```

6.3.2 File/Device Format Attribute

In the OPEN statement, you can indicate file or device format by the keywords DISPLAY or INTERNAL. Data is stored in files in either format, depending on the type of file required (sequential, relative, or key-indexed). You must use DISPLAY format with devices.

6.3.2.1 DISPLAY Format. The display specification indicates that the file is to be stored in ASCII format; that is, each character is written to the file as an ASCII code. This format is especially useful when the files are to be transferred to other systems. DISPLAY is the default for the OPEN statement and must be used in conjunction with devices and sequential files or KIFs.

The following examples, two equivalent statements, open a sequential file DS01.MYFILE on unit number 1. (Since the default file organization is sequential and the default format is DISPLAY, you need not specify these attributes.) The format of the file is ASCII. Records written to the file are written as ASCII codes.

EXAMPLES

```
OPEN #1:"DS01.MYFILE",SEQUENTIAL,DISPLAY
```

```
OPEN #1:"DS01.MYFILE"
```

6.3.2.2 INTERNAL Format. The internal specification must be specified when opening or creating relative record files. INTERNAL indicates that the data is stored in internal memory image format. In this format, each numeric value is written to the record in a format that depends on its type definition in the program (REAL, DECIMAL, INTEGER, or string). The amount of memory used to store the data depends on the data type and the data itself. Table 6-1 shows the memory requirements for each data type.

Table 6-1. Memory Requirements for INTERNAL Format Data

Data Type	Memory Requirement
String	1 byte per character + 1 byte
Integer	2 bytes
Real	8 bytes
Decimal	8 bytes

A numeric value is stored as an integer if it can be accurately represented; otherwise, it is stored as a real constant.

The following example opens the relative record file DS01.MYFILE on unit number 1.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",RELATIVE,INTERNAL,FIXED
```

6.3.3 File/Device Record Length Attribute

Each open file has a record length attribute that specifies whether the records in the file are fixed or variable in length. Sequential files, KIFs, and devices have variable record lengths. Relative record files have fixed record lengths. The keywords VARIABLE and FIXED are included in the OPEN statement to specify this attribute.

You should specify the length attribute in the OPEN statement when creating a sequential or relative record file.

6.3.3.1 Variable Record Length. The variable record length attribute specifies that the records in a file or for a device can be of varying lengths. If the OPEN statement does not specify a record length type, the variable record type is assumed. You can specify with an integer expression to the right of the keyword an optional maximum length in bytes. If you do not specify a maximum length, a default maximum length of 80 characters for files and the standard line size for devices is assumed.

If data exceeds the record length specified, the current record is terminated and the remaining data is written in the next record. If you do not specify record type, variable record type is assumed. The variable attribute can be used only with sequential file structures, KIFs, or devices.

In the first example that follows, the OPEN statement creates a sequential file named DS01.MYFILE on unit number 1. The file can contain at least 500 records, the internal format is ASCII, and the records are variable in length with a maximum length of 100 bytes. In the second example, the OPEN statement opens the device LP01 on unit 7 with a variable record length of 132 bytes (characters).

EXAMPLES

```
OPEN #1:"DS01.MYFILE",SEQUENTIAL 500,DISPLAY,VARIABLE 100,OUTPUT
```

```
OPEN #7:"LP01",VARIABLE 132
```

6.3.3.2 Fixed Record Length. The fixed attribute specifies that record lengths are of a fixed size. You can specify the size of a file by including an integer expression following the keyword FIXED. An attempt to read or write a record that exceeds the specified length results in an error. A relative record file requires fixed length records. You must specify the fixed attribute when opening a relative record file, and the record length specified must match the record length specified for the file when it was created.

The following example creates a relative record file named DS01.MYFILE on unit number 1. The data written to the file is in internal format, and the fixed length of each record is 256 bytes.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",RELATIVE,INTERNAL,FIXED 256,OUTPUT
```

6.3.3.3 Physical Record Length. A physical record is the size of a block of data in the files. Taking physical record length into consideration during file creation can optimize file management.

The default physical record length is operating-system dependent. If the logical record length of your file exceeds the default, use the Create File (CF) SCI command to create a new file. Specify a larger physical record length, one that allows several logical records to be stored in one physical record, improving system throughput. See your operating system manual for details on file creation.

6.3.4 File Life

Files created during a BASIC session are regarded as permanent unless explicitly deleted before the end of program execution. BASIC allows you to specify whether the file is to be permanent or temporary when it is created by specifying the keyword PERMANENT or TEMPORARY in the OPEN statement. The default is PERMANENT. If a file is specified as temporary, the file is deleted when the CLOSE or BYE statement is executed, or when another program is run. The file in the following example would be deleted upon closing.

EXAMPLE

```
OPEN #1:"DS01.SCRATCH", OUTPUT, TEMPORARY
```

NOTE

Not all operating systems support temporary files. See Appendix E, BASIC System Differences, for details.

WARNING

BASIC will allow the creation of a temporary file with the same file name as an existing file name on disk. If the temporary file is closed with the "delete" option, both the temporary file and the permanent file are deleted.

6.3.5 File Access Attribute

You can specify one or more access modes to instruct BASIC which operations can be performed on a file. These specifications include creating a new file and specifying read-only access, write-only access, or both read and write access to a file.

6.3.5.1 Output Access Mode. You must specify OUTPUT in the OPEN statement if you want BASIC to create the file. You cannot do this in opening a device, since BASIC cannot create a device (for example, LP01). Likewise, you cannot specify the output mode in the OPEN statement for KIFs, since they must be created in SCI mode. In the following example, the file DS01.MYFILE is created and opened on unit number 1. The file access mode is write-only. Since other file attributes are not stated, the defaults apply; the result is a sequential file in display format with variable length records whose maximum length is 80 characters.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",OUTPUT
```

6.3.5.2 Input Access Mode. The input mode specifies that a file can be read. In response to an attempt to write a file opened with only the input mode specified, an error message appears and program execution halts. The input option can be combined with the output option. Using this combination, a new file is created with both read and write access.

The following example opens a file named DS01.MYFILE on unit number 1. The file can only be read.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",INPUT,RELATIVE 50,FIXED 200,INTERNAL
```

The following example creates and opens a file named DS01.MYFILE on unit number 1. The program can read records from the file or write records to the file.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",OUTPUT,INPUT
```

6.3.5.3 Update Access Mode. The update mode specifies that a file can be both read and written, and serves as the default file access mode.

In the following example, the file DS01.MYFILE is opened on unit number 1. The program can read records from the file or write records to the file.

EXAMPLE

```
OPEN #1: "DS01.MYFILE",UPDATE
```

6.3.5.4 Append Access Mode. You can use the append mode only with the sequential file structure. This mode indicates that records can be written to the end of the file, but no records can be read. When the file is opened, the file position indicator is positioned past the last record in the file so that subsequent writes are to the end of the file. If a read operation is attempted, an error occurs and program execution halts. If a RESTORE statement is executed, a file mode error results.

The following example opens a file named DS01.MYFILE on unit number 1. Records can only be written or appended to the file.

EXAMPLE

```
OPEN #1:"DS01.MYFILE",APPEND
```

6.4 CLOSE STATEMENT

The CLOSE statement disassociates a file or I/O device from a unit number. After execution of the CLOSE statement, the file or device is inaccessible to the program and the unit number is available for reassignment. If the unit number specified is not associated with an open file, an error condition results. The general form of the CLOSE statement is as follows:

FORMAT

```
CLOSE # unit__number
```

```
CLOSE # unit__number: DELETE
```

where:

unit__number is any numeric value or expression greater than 0 and less than 256.

All open files and devices are automatically closed upon execution of a NEW, OLD, or RUN command.

BASIC supports a CLOSE with DELETE function. To delete a file opened with the OPEN statement, add :DELETE to the CLOSE statement. If this statement is addressed to a device, it merely closes the device. The following statements delete the file .JUNK on disk DS01:

EXAMPLES

```
OPEN #1:"DS01.JUNK"  
CLOSE #1:DELETE
```

6.5 PRINT AND DISPLAY STATEMENTS

The PRINT and DISPLAY statements format and transmit the output of the BASIC program to a disk file or output device. The first two format statements describe the PRINT statement used for screen output; the others are used for output to files and devices in general.

FORMATS

PRINT options: output__list

PRINT options USING image: output__list

PRINT # unit__number: output__list

PRINT # unit__number USING image: output__list

PRINT # unit__number, rec__clause: output__list

PRINT # unit__number, key__clause: output__list

PRINT # unit__number, key__clause USING__image: output__list

where:

options indicates ERASE ALL, AT (row, col), SIZE (int), or BELL.

unit__number is any numeric value or expression greater than 0 and less than 256 that specifies the disk file or output device.

image is the line number of an IMAGE statement or a string expression.

output__list specifies the data items to be printed. The data items can be numeric variables, strings, or expressions that are evaluated and then printed. The output list can also contain column and position information.

rec__clause location of the record to be written; consists of the word REC followed by any legal arithmetic expression.

key__clause location of the record to be written; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or by a key value.

The DISPLAY statement has the same syntax as the PRINT statement. You can use the DISPLAY and PRINT statements interchangeably when sending data to the output devices or files, except that if you specify screen options with the PRINT statement, these options are ignored if you direct output to a device other than the screen.

In the remainder of this section, the term *PRINT statement* describes both the PRINT and DISPLAY statements. Aside from the exception noted above, the options available with the PRINT statement apply equally to the DISPLAY statement. Many extensions are available to both statements, making them two of the most useful statements in the BASIC language.

6.5.1 Device Output

To send output to a device, use the PRINT or DISPLAY statement with a unit number that specifies the device. All devices accessed with a unit number other than zero must be opened by using the OPEN statement before the output statement can be executed. The simplest forms of the PRINT statement are as follows:

EXAMPLES

```
PRINT
```

```
PRINT #1
```

Execution of the preceding PRINT statements advances the output position to the beginning of the next line.

Numeric values are displayed with a leading sign position (blank if positive and a minus sign if negative) and a trailing blank. String values are displayed without these additions.

The following is an example of a PRINT statement that outputs the value of an expression.

EXAMPLE

```
100 PRINT "TEST DATA"
110 PRINT 4 + 8
120 PRINT 4-8
```

```
RUN
```

```
TEST DATA
12
-4
```

6.5.2 Data Separators

You can include multiple data items in a single PRINT statement if you separate them by data separator symbols. BASIC provides three data separator symbols for use within the PRINT statement: the comma, the semicolon, and the apostrophe. The effect of these separators is independent of whether the data is numeric or string. If a list ends with a data separator, the output position does not advance to the next line after the values of the expressions in the list are printed; consequently, the results from more than one DISPLAY statement can appear on a single line. Except for this attribute, each data separator acts differently, as explained in the following paragraphs.

6.5.2.1 Comma. For DISPLAY format output (sequential files, KIF, and devices), the comma data separator advances the output display position to the next zone. The output line is divided into zones; each zone is 16 spaces wide. The following example shows the PRINT statement that uses a list and the comma data separator.

EXAMPLE

```
110 PRINT 1,2,3  
  
RUN  
  
1           2           3
```

You can achieve the same result by appending a comma to separate PRINT statements, as in the following example.

EXAMPLE

```
110 PRINT 1,  
120 PRINT 2,  
130 PRINT 3  
  
RUN  
  
1           2           3
```

For internal format output (relative record files), the comma causes no formatting. Data items are output one after the other to the relative record file.

6.5.2.2 Semicolon. When you use the semicolon data separator, BASIC does not advance the output position after displaying a value. If a semicolon replaces the comma in the preceding example, no space appears between the data fields.

EXAMPLE

```
100 DISPLAY 1;2;3  
110 DISPLAY "ABCD";"EFGH"  
  
RUN  
  
1 2 3  
ABCDEF GH
```

In this example, the blanks between 1 and 2 and the blanks between 2 and 3 represent the sign position and the trailing blank; the semicolon does not cause any blanks.

6.5.2.3 Apostrophe. The apostrophe data separator has the same effect as the semicolon except that the data separator symbol (the comma unless otherwise defined by the PUNCTUATION statement) is inserted between the data items. This is particularly useful in transmitting data to files. See paragraph 6.10 for special considerations in using the apostrophe separator with KIFs.

The following example shows the different effects of the three data separators.

EXAMPLE

```

100 DISPLAY 1,2,3
110 DISPLAY 1;2;3
120 DISPLAY 1^2^3
130 DISPLAY "HIGH";"LOW";"DEEP"
140 DISPLAY "HIGH" ^ "LOW" ^ "DEEP"
150 DISPLAY 1;
160 DISPLAY 2^
170 DISPLAY 3
180 END

```

RUN

```

1           2           3
1  2  3
1 , 2 , 3
HIGHLOWDEEP
HIGH,LOW,DEEP
1  2  ,  3

```

6.5.3 Output Options

The following options are available with the PRINT and DISPLAY statements: ERASE ALL, AT, SIZE, and BELL. You can use these options only when the format of the output statement does not include a unit number. When you use any of the options, you must precede the output list by a colon. When you use two or more options in combination, you must list them in the order previously given.

6.5.3.1 PRINT with ERASE ALL Option. The ERASE ALL option clears the screen before any values are displayed.

FORMAT

ERASE ALL

The following is an example of a PRINT statement that uses the ERASE ALL option. If you select this option, it must immediately follow the keyword PRINT.

EXAMPLE

```
PRINT ERASE ALL:"Total = ";SUM
```

6.5.3.2 PRINT with AT Option. The AT option specifies the starting position for the display on the screen.

FORMAT

AT (line__num, col__num)

The default value for the line is 24, the bottom line; the default value for the column is column 1. If you do not specify the AT option and no data separators are in effect, a DISPLAY statement causes output to begin in the lower left corner of the screen (the default position). Therefore, if a PRINT statement without the AT option follows a PRINT statement with one, the output of the second statement begins in the lower left corner, regardless of where the cursor was positioned by the AT option in the previous statement. If the AT and ERASE ALL options are both used in the same PRINT statement, the keyword AT must follow the keywords ERASE ALL. The following example shows a PRINT statement that uses the AT option.

EXAMPLE

```
PRINT AT(10,1):XNUM
```

6.5.3.3 PRINT with SIZE Option. The SIZE option has the following form:

FORMAT

SIZE(n)

where:

n equals the number of characters to be displayed (can be positive or negative).

The SIZE option declares the maximum number of characters that can be displayed. If you do not specify a size, the size defaults to a value large enough to hold the data to be output plus the number of characters to the end of the last line on which data is written. When a line is displayed on the screen, it is cleared according to the size specification.

You can use the **SIZE** option to display data to a line and to preserve previously displayed data in the line after the field specified by the size option. The field is cleared for the length of the size specification only. If a string is to be displayed with a length greater than the size specification, the output is truncated on the right after the number of characters in the **SIZE** specification has been displayed. If you specify a negative size, the absolute value is used and no error occurs. If the **AT** option is also present, it must precede the **SIZE** option as shown in the following example:

EXAMPLE

```
PRINT AT (20,10) SIZE(21): "THIS IS 21 CHARACTERS"
```

6.5.3.4 PRINT with BELL Option. The **BELL** option sounds the bell when the **PRINT** statement executes. You can achieve the same result by using **CHR\$(7)**.

FORMAT

BELL

Using the **BELL** clause does not affect the format or interpretation of any data that the statement is processing. If you use the **BELL** option with **ERASE ALL**, **AT**, or **SIZE** options in the same **PRINT** statement, the keyword **BELL** must follow the others.

EXAMPLE

```
PRINT AT (20,10) SIZE(19) BELL: "THIS RINGS THE BELL"
```

6.5.4 PRINT with USING Option

The **USING** option can control output format. The form of the option is either of the following:

FORMATS

USING line__num

USING str__exp

where:

line__num is the line number of an **IMAGE** statement.

str__exp contains the format image.

PRINT statements that include the USING option can contain the same elements as a PRINT list except that the expressions in the list must be separated by commas and only a semicolon can be used as a trailing data separator. The keyword USING must follow all of the other option keywords in a PRINT statement. The following is an example of the USING option.

EXAMPLE

```
10 VALUE = 1234.569
20 A$ = "####.##"
30 IMAGE ####.##
40 PRINT USING "####.##":VALUE
50 PRINT USING A$:VALUE
60 PRINT USING 30:VALUE
```

RUN

```
1234.57
1234.57
1234.57
```

The preceding PRINT USING statements all produce the same output, 1234.57. Since the interaction between the print list and the image are the same whether they are contained in a string expression or in an IMAGE statement, further discussion is in terms of the IMAGE statement form only.

If the print list includes more values than there are conversions specified in the associated IMAGE statement, the excess values are printed on a new line and the IMAGE statement is reevaluated, beginning with the first character. If fewer values are to be printed than the IMAGE statement provides for, the printing terminates when the first data conversion field for which no value is encountered.

6.5.5 IMAGE Statement

The IMAGE statement is used only in association with a USING clause in a PRINT or DISPLAY statement; its line number is specified in the clause. The IMAGE statement provides the template used in formatting values in the PRINT statement list.

FORMAT

IMAGE str__const

where:

str__const is a quoted or unquoted string constant.

The string constant contains directions for formatting the values in the display list as well as text to be included in the list. The text characters in the output line appear in the same positions as in the string constant. Text characters are any characters not specified as format control characters.

6.5.5.1 IMAGE Format Control Characters. Table 6-2 lists the nine format control characters and their corresponding functions.

Table 6-2. Format Control Characters

Character	Function
Pound sign (#)	Serves as the replacement field for a data character
Circumflexes (AAA)	Provide positions for displaying the exponent. If more positions are provided than are required by the exponent, leading zeros are displayed.
Plus sign (+)	When placed at the beginning of a field, it causes a floating sign to be displayed to the left of a number.
Decimal point (.)	Indicates the position of the decimal point symbol. It can also be used to align the decimal points in a column of numbers.
Angle brackets (< >)	When a numeric conversion field is enclosed by angle brackets, the brackets are displayed when the value of the field is negative.
Comma (,)	Used in a numeric field to insert digit separator characters in specified output positions.
Dollar sign (\$)	Causes a currency symbol to be displayed at the beginning of the specified field. To obtain a floating currency symbol, use two dollar signs.
Asterisks (**)	When specified in pairs at the beginning of a numeric conversion, causes leading zeroes to be replaced by asterisks, providing blank protection.

When encountered in an image, any characters other than those in Table 6-2 are reproduced in the output without editing. The following example shows statements that use format control characters.

EXAMPLE

```
100 A = 123.546
110 B = 24.68
120 C$ = "AUSTIN"
130 IMAGE ##### TI BASIC < ###.#>

140 FORMAT$ = "#.####^AAA $$###.##"
150 PRINT USING 130:A,B
160 PRINT USING FORMAT$:A,B
170 PRINT USING 130:C$,-B
180 PRINT USING " + ### #####.#":A,-B
```

RUN

```
124 TI BASIC 24.7
.1235E + 03 $24.68
AUSTIN TI BASIC < 24.7>
+ 124 -24.7
```

The format control characters used to represent the different types of data are controlled by the rules discussed in the following paragraphs.

6.5.5.2 Integer Fields. An integer field is composed of digits with an optional sign. If the specified number overflows the field, asterisks are displayed instead of the value. Numerical data is right justified and rounded. The sign of the number is included in the number of digits. A maximum of 14 significant digits can be displayed.

The following example uses the IMAGE option with integer fields. Note that the last variable (C = 1289.9999) overflows the image specification (###); thus, its output field is filled with asterisks.

EXAMPLE

```
100 IMAGE ##### ##### ###
105 A = 123.45 :: B = -34.856 :: C = 1289.9999
120 PRINT USING 100: A,B,C
130 END
```

RUN

```
123 -35 ***
```


The following example shows IMAGE assigned to a string variable.

EXAMPLE

```

100 A$ = "#### ##### ###"
110 A = 123.45::B = -34.856::C = 1289.999
120 DISPLAY USING A$:A,B,C
130 END

```

RUN

```

123  -35 ***

```

6.5.5.3 Decimal Fields. A decimal field is a string of pound signs with an optional leading sign and a decimal point that may precede, be embedded in, or terminate the field. The specified number is rounded to the number of places indicated by the pound signs that follow the decimal point. The number is right justified, and the decimal point is placed in the position specified in the field definition. When the number overflows the field, asterisks are displayed instead of the value. Up to 14 significant digits can be displayed. A display field that contains more than 14 numeric characters causes a program error.

The following example uses the IMAGE option with decimal fields.

EXAMPLE

```

100 IMAGE ####.## #####.##### #####. #.###
110 LET A = 123.456 :: B = -34.856 :: C = 47.7 :: D = -.0177
120 PRINT USING 100: A,B,C,D
160 END

```

RUN

```

123.46    -34.8560    48.    -.018

```

6.5.5.4 Exponential Fields. An exponential field is a decimal or integer field followed by four or five circumflexes (depending on the size of the exponent), which reserve a place for the exponent. Fewer than four or more than five are treated as a literal string. The specified number is rounded in the same manner as in decimal fields. The leftmost unary sign (-, +) reserves a position for the sign of the number: minus if negative, blank if positive. At least one field character (#, -, +) must precede the period in the image. Up to 14 significant digits can be displayed.

The following example uses the IMAGE option with exponential fields.

EXAMPLE

```
100 IMAGE #.#####^AAA ##.###^AAA ###.^^A #.##^AAA
110 A = 123.456 :: B = -34.856 :: C = 47.7 :: D = -.0177
120 PRINT USING 100:A,B,C,D
160 END
```

RUN

```
.12346E+03 -3.486E+01 48.E+00 -.18E-01
```

6.5.5.5 Alphanumeric Fields. The number sign or any other format control character is used to indicate the position and length of the field. If the string is shorter than the format specification, the field is filled with blanks on the right. If the string is longer than the format specification, the output field is filled with asterisks.

The following example uses the IMAGE option with alphanumeric fields. Note that the second variable (B\$ = "ABCDEFGH") is larger than the image specification (#####); thus, asterisks fill the second output field.

EXAMPLE

```
100 IMAGE #####
110 LET A$ = "ABC" :: B$ = "ABCDEFGH"
120 PRINT USING 100: A$,B$
130 END
```

RUN

```
ABC      *****
```

6.5.5.6 Literal Fields. A literal field is composed of characters that are not format control characters. A literal field appears on the line exactly as it appears in the image. For example, the image string ABCDEF consists of a six-character literal field.

EXAMPLE

```
100 PRINT USING "THE TOTAL IS ###":503
```

```
RUN
```

```
THE TOTAL IS 503
```

6.5.6 File Output

The BASIC program uses the PRINT statement to transmit data to a file. It can send data to sequential, relative record, and key-indexed files. To initiate output, open the file by using the appropriate OPEN statement. To send data to the file, specify the file's unit number in the PRINT statement.

6.5.6.1 Sequential File Output. The display format, used with the sequential file structure, forces the data written to a record to appear in the same format as the data written to your screen. In the following example, the PRINT statement outputs four variables (A, B, C, and D) to the sequential file associated with unit number 1. The comma data separator inserts spaces in the record to position the variables at the beginning of output zones. If the file specification is removed from the PRINT statement or if unit number 0 is used, the values are written to your screen in exactly the same format.

EXAMPLE

```
PRINT #1:A,B,C,D
```

```
1          2          3          4
```

The following statement can also output data to a sequential file:

```
PRINT #1:A'B'C'D
```

However, this statement produces a record containing the values of variables A, B, C, and D separated by data separators (in this case, commas), as follows:

```
1, 2, 3, 4
```

String data is treated in the same manner, as indicated by the following example.

EXAMPLE

```
100 NAMES$ = "DOE,JOHN"  
110 ADDRESS$ = "123 BARTON SPRINGS RD"  
120 PRINT #1:NAMES$'ADDRESS$
```

Executing this statement sends the following output to a sequential file record:

```
DOE,JOHN,123 BARTON SPRINGS RD
```

To output a large number of variables (more than could be contained in a single variable list) on a single record, terminate the last variable in the variable list with a data separator symbol (' , ;). This causes data to be sequentially output to a record until that record is filled, as in the following example.

EXAMPLE

```
100 FOR I= 1 TO 40  
110 PRINT #1:DAT(I)  
120 NEXT I  
130 PRINT #1
```

The apostrophe data separator following the PRINT statement informs BASIC that more values are to be written to the current record. The final PRINT statement at line 130 indicates that the record is complete, and the next PRINT statement to unit number 1 begins a new record. The apostrophe data separator inserts a comma between each value in the output record.

6.5.6.2 Relative Record File Output. The INTERNAL format, always used with relative record files, produces data records with binary values equivalent to their specified data type (REAL, INTEGER, or DECIMAL). Data separators do not cause formatting; data items are output one after the other to the relative record file.

The PRINT statement can specify the record number (REC) by using the REC clause when sending output to a relative record file. The REC clause can contain any legal arithmetic expression.

In the following example, the PRINT statement writes data to record 1 of the relative record file, DS01.RELFILE.

EXAMPLE

```
110 PRINT #1,REC 1:A,B,C,D
```

If the variables A, B, C, and D are defined as integers, the PRINT statement produces a record of eight bytes, as follows: bytes 1 and 2 contain the value of variable A, bytes 3 and 4 contain the value of variable B, and so on. The data is stored in REC 1 of the file DS01.RELFILE without any data separators. If the fixed record length is specified as less than the number of characters or bytes required to write the record, the data is continued on the next record.

If string data is written to a relative record file, the length of the string determines the number of characters written. The number is equal to the length plus one. The following example illustrates output of string values to a relative record file.

EXAMPLE

```
110 NAME$ = "ABC"
120 ADDR$ = "DEF"
130 PRINT #1,REC 1:NAME$,ADDR$
```

In the preceding example, the PRINT statement writes eight characters or bytes to record number 1 of a relative record file, that is, one byte for the length of NAME\$ followed by the three-byte string NAME\$ and one byte for the length of ADDR\$ and the three-byte string ADDR\$.

To output a large number of variables (more than could be contained in a single variable list) on a single record, terminate the last variable in the variable list with a data separator symbol (',;'). This causes data to be sequentially output to a record until that record is filled, as in the example below.

When a long record is written to a relative record file, ensure that the REC clause is specified only in the first write to the record. The occurrence of the REC clause in an I/O statement causes a new record to be accessed. In the following example, 40 values are written to record N of a relative record file.

EXAMPLE

```
100 PRINT #1,REC N:DAT(1),
110 FOR I = 2 TO 40
120 PRINT #1:DAT(I),
130 NEXT I
140 PRINT #1
```

The initial PRINT statement must be outside the FOR loop so that the REC clause is specified only once.

6.5.6.3 KIF Output. The PRINT statement inserts a new record into a file. The KEY clause is ignored as far as the inserted record is concerned. The position of the inserted record depends on the value of its keys (defined in the record). The KEY clause does, however, specify a particular key number so that a subsequent INPUT statement to the file with no KEY clause specified will read the record following the one inserted.

In KEY files, the data and key values should be padded to the right with blanks when necessary to fill their fields prior to the PRINT statement. The record specified in the PRINT statement must be large enough to contain all key values; KIFs are variable in record length, with a minimum as well as a maximum. If the record is too small, an error occurs and program execution halts.

The following KIF example shows a file that has one key with a maximum length of 20 characters specified in position 1. However, the logical record length is specified as 50 bytes to allow room for data. Note that the SIZE specification on the ACCEPT statement prevents entering values larger than the maximum fields defined.

EXAMPLE

```

100 ACCEPT SIZE(20):CITY$
110 IF LEN(CITY$)> 19 THEN 140 ! NO PAD NECESSARY
120 ! PAD KEY TO MAXIMUM LENGTH
130 CITY$ = CITY$ & RPT$(" ",20-LEN(CITY$))
140 ACCEPT SIZE(20):STATES$
150 PRINT #1:CITY$'STATES$

```

When a long record is written to a KIF, the same conventions apply as when a long record is written with a REC clause in a relative record file. Including a KEY clause terminates a record. In the following example, 20 string values are written to a KIF, where character positions 1 through 5 have been defined as the key.

```

100 K$ = "ABCDE"
110 PRINT #1:K$'
120 FOR I= 1 TO 19
130 PRINT #1:DATA$(I)'
140 NEXT I
150 PRINT #1

```

A record is inserted with the primary key equal to ABCDE, followed by 19 data values.

6.5.7 REPRINT Statement

The REPRINT statement allows you to update a KIF record. Updating a record entails reading, modifying, and rewriting the record. To rewrite a record, you must have locked it by a previous INPUT or ACCEPT statement. REPRINT updates and unlocks the record.

FORMATS

REPRINT # unit__number

REPRINT # unit__number, key__clause

where:

- | | |
|--------------|---|
| unit__number | is any numeric value or expression greater than 0 and less than 256 that specifies the KIF. |
| key__clause | location of the record to be rewritten; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or key value. |

The REPRINT statement specifies the unit number attached to the file, optionally followed by a KEY clause. The KEY clause defines the location of the record to be rewritten. If you omit the KEY clause, the default value of the record is the record designated by the last operation on the file. The following is an example of the REPRINT statement:

EXAMPLE

```

100 DIM DATA$(20)
110 OPEN #1:".KEYFILE",KEYED
120 INPUT #1, KEY #2 "CASSIUS CLAY"           ", LOCK:SSN$, NAME$,
130 IF EOF(1) THEN 250
140 FOR I = 1 TO 19
150 INPUT #1: DATA$(I),
160 NEXT I
165 INPUT #1: DATA$(20)
170 IF NAME$ <> "CASSIUS CLAY" THEN UNLOCK #1 :: GOTO 250
180 IF LEN(SSN$) >= 9 THEN 200
190 SSN$ = SSN$ & RPT$(" ",9-LEN(SSN$))      ! BLANK FILL
200 REPRINT #1: SSN$^"MOHAMMED ALI          " ^! BLANK FILL
210 FOR I = 1 TO 20
220 REPRINT #1: DATA$(I)^
230 NEXT I
240 REPRINT #1
250 CLOSE #1
260 STOP

```

In this example, if a client named Cassius Clay had changed his name to Mohammed Ali, you could update the record accordingly. During the creation of this KIF, social security number (SSN) was designated key 1, starting at position 1, with a length of 9. Key 2 (NAME) starts at position 11 (which leaves room for a data separator between keys 1 and 2); its length is 25.

6.5.8 SCRATCH Statement

The SCRATCH statement enables you to delete records within a KIF.

FORMATS

SCRATCH # unit__number

SCRATCH # unit__number, key__clause

where:

unit__number is any numeric value or expression greater than 0 and less than 256 that specifies the KIF.

key__clause location of the record to be deleted; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or key value.

The SCRATCH statement specifies the unit number of the file being edited, optionally followed by a KEY clause. If you omit the KEY clause, the default value is the next record in sequence. The following example demonstrates the SCRATCH statement:

EXAMPLE

```
300 SCRATCH #1,KEY #5 "TEXAS"  
500 SCRATCH #7
```

In the first line, the first record whose fifth key value is equal to TEXAS is deleted. In the second line, the record to which the data pointer is currently positioned via a RESTORE or other I/O statement is deleted.

WARNING

The SCRATCH statement scratches the next key greater than or equal to the specified key. If the key does not exist, the next higher key is scratched.

6.6 INPUT AND ACCEPT STATEMENTS

The INPUT and ACCEPT statements can accept data from the keyboard during program execution and receive data from files. The statements differ in that INPUT can include a list similar to a PRINT statement, but ACCEPT can receive only a single variable. The ACCEPT statement receives its single variable with no editing for data separators.

6.6.1 INPUT Statement

The INPUT statement assigns values obtained from either the keyboard or a file to variables within the program. The INPUT statement has two forms: one for screen input and another (the general form) for device and file input.

FORMATS

INPUT options prompt: variable__list

INPUT # unit__number: variable__list

INPUT # unit__number, rec__clause, lock__clause: variable__list

INPUT # unit__number, key__clause, lock__clause: variable__list

where:

options	specifies ERASE ALL, AT, SIZE, or BELL.
prompt	specifies an optional message that is printed immediately in front of the input field on the screen.
variable__list	specifies the data items to be entered. When the list specifies more than one variable, the variables must be separated by commas.
unit__number	specifies the file or device from which the data is to be entered. If no unit number is specified, the keyboard is the input device.
rec__clause	location of the record to be read; consists of the word REC followed by any legal arithmetic expression.
lock__clause	consists of the optional keyword LOCK.
key__clause	location of the record to be read; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or by a key value.

6.6.2 Keyboard Input

You can use the INPUT statement to enter data from the keyboard. No unit number need be specified. If the variable is a string variable, the data entered is interpreted as string data; if the variable is numeric, enter numeric data. The following are several examples of valid INPUT statements that enter data items from the keyboard.

EXAMPLES

```
INPUT PAY__RATE
```

```
INPUT POWER,WEIGHT(I),TESTER$
```

```
INPUT REC__NO,DESC$
```

```
INPUT BELL "Enter the value" : X
```

When the INPUT statement is executed, you are prompted to enter a value. All data entered from the keyboard in response to a single INPUT statement must be contained on one line of the screen. Leading and trailing blanks are removed from each data item upon input. Separate multiple variable values by commas. Use the comma to separate one data item from another upon input. You can use the PUNCTUATION statement to change the data separator symbol from a comma to another character.

Options available with the INPUT statement are selected the same way as DISPLAY statement options. When you use an option, you must precede the variable list with a colon.

6.6.2.1 INPUT with ERASE ALL Option. You can use the ERASE ALL option with the INPUT statement. This option erases the screen before the remainder of the statement is processed. If you select it, you must include it immediately after INPUT. The following is an example of the INPUT statement using the ERASE ALL option.

EXAMPLE

```
INPUT ERASE ALL:HOURS
```

6.6.2.2 INPUT with AT Option. You can use the AT option with INPUT in the same fashion as with the DISPLAY statement. The cursor moves to the specified position, and the prompt is issued. Keyboard data is accepted after the prompt appears. The following example demonstrates the use of the AT option.

EXAMPLE

```
INPUT AT(5,9*J + 2)“QUANTITY”:QUAN(J)
```

If you use both the AT and ERASE ALL options in the same INPUT statement, the keyword AT must follow the keywords ERASE ALL.

6.6.2.3 INPUT with SIZE Option. You can use the SIZE option with the INPUT statement. The absolute value of the size specification declares the maximum number of characters that you can enter. If you attempt to enter more characters than the size specification allows, the bell sounds. If you do not enter a size specification, the field size defaults to the remainder of the line following the input prompt.

If the size specification is positive, the data entry area is cleared before data is accepted. If the size specification is negative, the data entry area is not cleared and you can provide default values. This simplifies data entry when the data usually, but not always, has a fixed value (for example, country of customer). The following example uses the size specification with a negative argument.

EXAMPLE

```
10  DEFAULT = 12
20  DISPLAY ERASE ALL AT (12,44): DEFAULT
30  INPUT AT (12,25) SIZE (-2), “Number of doughnuts-”:DOUGHNUTS
```

Executing this example causes the following to appear:

```
Number of doughnuts-12
```

If the typical number of doughnuts is ordered (in this example, 12), you need only press the carriage return. However, if seven doughnuts are ordered, you must enter a 7, followed by a blank. The blank is required to erase the second digit of the default value. If the size specification had been positive, the data entry area would be erased within the bounds of the size specification, prior to the input prompt, thereby eliminating the default INPUT value. If the AT option is also present, it must precede the SIZE option, as shown in the following examples.

EXAMPLES

```
INPUT AT (10,10) SIZE(3),“ENTER 3 LETTERS”:A$
```

```
INPUT ERASE ALL AT (5,5) SIZE(5):B$
```

If the SIZE and ERASE ALL or the SIZE and AT options are used in the same INPUT statement, the keyword SIZE must follow the keywords ERASE ALL or AT.

6.6.2.4 INPUT with BELL Option. The BELL option causes the bell to ring when the INPUT statement executes. The form of this option is as follows:

FORMAT

BELL

Using the BELL option does not affect the format or interpretation of any data that the statement is processing.

EXAMPLE

```
INPUT AT (10,10) SIZE(3) BELL, "IF THE BELL RINGS, ENTER YES":A$
```

If you use the BELL option with the ERASE ALL, AT, or SIZE options in the same INPUT statement, the keyword BELL must follow the keywords ERASE ALL, AT, or SIZE.

6.6.2.5 Input Prompting. If a prompt is not explicitly included in the INPUT statement, the default prompt, which is a question mark followed by a space (?), appears on the screen. If a more descriptive prompt is needed, you must include a prompt clause in the INPUT statement. The clause is a string expression and is displayed instead of the question mark and blank. You can use the empty string, "", as the prompt clause if no prompting is needed. The following provides an example of prompting for input.

EXAMPLE

```
10 INPUT "NUMBER PLEASE ":N
20 DISPLAY "THE SQUARE ROOT OF";N; "IS";SQR(N)
30 INPUT "MORE INPUT?" :A$
40 IF A$ = "NO" THEN 60
50 GO TO 10
60 DISPLAY "THANK YOU, GOODBYE"
70 END
```

RUN

```
NUMBER PLEASE 16
THE SQUARE ROOT OF 16 IS 4
MORE INPUT? YES
NUMBER PLEASE 25
THE SQUARE ROOT OF 25 IS 5
MORE INPUT? NO
THANK YOU, GOODBYE
```

6.6.2.6 Input Errors. A number of different errors can result from improper use of the INPUT statement. The following indicates the most common input errors.

Error	Meaning
80	Too much input data
83	Too little input data
2	Missing or mistyped number

You can handle these errors by using the special error trapping statements (for example, ON ERROR) or you can allow the system to indicate the error. When the system indicates an input data error, an error code appears on the screen for several seconds. The INPUT statement is then reexecuted.

6.6.3 INPUT Statement in File Access

You can use the INPUT statement to receive data from sequential and relative record files. Before you can use the INPUT statement to access a file, you must open the file with the appropriate OPEN statement. Also, file INPUT statements cannot use the prompt option or the screen options.

6.6.3.1 Sequential File Input. Input of data from a sequential file is similar to input from a device. When you input multiple values, you must separate them by the data separator symbol. Otherwise, they appear as one data item. Leading and trailing blanks are removed from each data item upon input. The following statements are examples:

```
OPEN #1: "DS01.MYFILE"
INPUT #1: A,B,C
```

These statements successfully input three values from the sequential file that contained data in the following format:

```
1,2,3
```

However, if the data is in the format 123, the contents of the record appear as a single value, 123; this value is assigned to the variable A. Subsequent records are then read until a sufficient amount of data is found to satisfy the variables B and C. If more data items are in a sequential record than the INPUT statement requires, only those needed to satisfy the INPUT statement are read. The remaining data items are discarded unless the INPUT statement has a data separator following the last variable name. In that case, the data remaining in the record is read by the next input operation.

6.6.3.2 Relative Record File Input. The INTERNAL format, always used with relative files, produces data records with binary values equivalent to their specified data type (REAL, INTEGER, or DECIMAL). Relative file I/O also allows the random access of records by specifying the record number to be read or written in the I/O statement.

In the following example, record number 1 is written to and then read from a relative record file. The record number specification can be any legal arithmetic expression. Whenever the REC clause is specified, a new record is read. If the variables A, B, C, and D are defined as integers, the PRINT statement produces an eight-byte record: bytes 1 and 2 equal the value of variable A, bytes 3 and 4 equal the value of variable B, and so on. Zones and data separators are not defined in a relative record file; input fields are defined by the types of variables in the INPUT variable list instead of the appearance of a comma data separator in the data record. Therefore, the apostrophe print separator cannot be used.

If the variables D, E, F, and G are defined as integers, the associated INPUT statement retrieves the values as written. However, if one or more of the variables in the INPUT list differ in type from those in the PRINT list, the results are unpredictable. If the fixed record length is specified as less than the number of characters or bytes required to write the record, an error results. Table 6-1 provides information on memory requirements for different data types stored in relative record files.

EXAMPLE

```
PRINT #1,REC 1:A,B,C,D
```

```
INPUT #1,REC 1:D,E,F,G
```

If string data is written to a relative record file, the length of the string determines the number of characters written.

To input multiple data items on a single line, use a data separator to terminate the INPUT statement. The occurrence of the REC clause in any I/O statement causes a new record to be accessed. In the following example, 40 values are read from record N of a relative record file.

EXAMPLE

```
100 INPUT #1,REC N:DAT(1),  
110 FOR I = 2 TO 40  
120 INPUT #1:DAT(I),  
130 NEXT I
```

6.6.3.3 KIF Input. To input data to a program from a KIF, use a KEY clause to identify the record you wish to access. A key clause consists of the keyword KEY followed, optionally, by a pound sign (#) and a key number and/or a key value.

FORMAT

statement # unit__number, KEY key__num, key__val: variable

where:

statement	represents a BASIC statement (for example, INPUT or ACCEPT).
unit__number	is the number assigned to the file in the OPEN statement.
key__num	is the pound sign (#) followed by the key number. If omitted, the primary key is used.
key__val	is the value of the key. If omitted, the next value on the specified key is used.
variable	specifies the data item to be entered.

If you do not specify the key number, the default value is the primary key. If you do not specify the key value, the next record in the current key is read. If the entire KEY clause is omitted, the next record in sequence is read.

EXAMPLE

```
INPUT #1,KEY#1 NAME$:NAM$,ADDR$,CITY$,STATE$
```

In this example, the INPUT statement specifies that four data values are to be read from a record whose first or primary key is equal to or greater than NAME\$. Note that if no record is found with a primary key of NAME\$, the record with the next greater primary key value is returned (if one exists).

The data format in a KIF is the same as the data format in a sequential file. When more than one value is stored in a record, a data separator symbol must separate each. Since keys are stored in the record with the data, the overall position of the information in the record is critical. To facilitate input, use the apostrophe separator when writing a record. Also, remember to leave space for the apostrophe separator during the key definition phase of file creation.

The following problem in file creation focuses on correctly positioning the apostrophe separator. During the creation of a KIF, two keys are specified. The first is defined as a 20-character string that begins in position 1 of the record. The second is a 20-character string positioned immediately after the primary key. Since the primary key occupies positions 1 through 20 and the apostrophe separator inserts a comma into position 21, the second key must begin in position 22. The following illustrates this record layout:

```
0           1           2           3           4
12345678901234567890123456789012345678901
----- KEY #1 -----,----- KEY #2 -----
```

Note that the second key occupies positions 22 through 41; therefore, the logical record length specified must be at least 42 characters (41 rounded up, since the length must be even).

The following example demonstrates how you can use KIF features to search a KIF. If a record is not found with a key value equal to the key value specified in the KEY clause, the record with the next greater key value is returned. Thus, you can search a KIF sequentially from a starting key number or value.

EXAMPLE

```
100 STATES$ = " "
200 INPUT #1,KEY #3 STATES$:NAME$,CITY$,STATES$
210 IF EOF(1) THEN 250
220 PRINT NAME$,CITY$,STATES$
230 INPUT#1:NAME$,CITY$,STATES$
240 IF EOF(1) = 0 THEN 220
250 CLOSE #1
260 END
```

In this example, the KIF #1 is searched sequentially on key number 3. Records are read and the data is printed until an end-of-file condition occurs.

6.6.4 ACCEPT Statement

The ACCEPT statement is like the INPUT statement with the following exceptions:

- You can input only a single variable.
- Trailing commas, semicolons, or apostrophes cannot follow the variable.
- The input data for an ACCEPT statement is not edited for commas or data separators. Commas can be part of the string data. Numeric data is handled as with the INPUT statement.
- The ACCEPT statement accepts all data on the input line (up to 255 characters) as the single-string data item.

In the following format description, the first format statement describes the ACCEPT statement used for screen input; the others are used for device and file input.

FORMATS

ACCEPT options prompt: variable

ACCEPT # unit__number: variable

ACCEPT # unit__number, rec__clause: variable

ACCEPT # unit__number, key__clause: variable

where:

options	indicates ERASE ALL, AT (row, col), SIZE (int), or BELL.
prompt	is the message displayed on the screen.
variable	specifies the data item to be used as input.
unit__number	is any numeric value or expression greater than 0 and less than 256 that specifies the file or device from which the data is to be entered. If you do not specify a unit number, the keyboard is the input device.
rec__clause	location of the record to be read; consists of the word REC followed by any legal arithmetic expression.
key__clause	location of the record to be read; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or key value.

The following are examples of an ACCEPT statement that uses some of the options available with the INPUT statement.

EXAMPLES

ACCEPT #1: A\$

ACCEPT "ENTER YOUR NAME" :N\$

ACCEPT AT(J,22) SIZE(4):PAY__GRADE

In input operations, the ACCEPT statement can read single data items from the keyboard and files. It accepts all of the data within the input record as a single-string data item, ignoring data separators. With this exception, it can be used in the same way as the INPUT statement.

6.7 PROGRAM DATA

Program data statements maintain constant data within a user program. The DATA, READ, and RESTORE statements manage and access the constant data.

6.7.1 DATA Statement

The DATA statement defines the data values that will be used in the program. The data values can be numeric or string constants. The program can contain several DATA statements; these statements need not be adjacent. As the program exhausts one DATA statement, BASIC scans lines with higher numbers to locate the next one.

FORMAT

DATA list

where:

list represents one or more numeric constants, or quoted or unquoted string constants, separated by commas.

The following is an example of the use of the DATA statement.

EXAMPLE

```
100 DATA 1,2,3
      .
      .
      .
170 DATA 4,5,6
180 DATA "Doe"
      .
      .
      .
220 DATA PARIS, ROME
230 DATA 4, CASH ,8.7 , " Title "
```

6.7.2 READ Statement

The READ statement assigns values to the variables listed in the READ statement by using data values in the DATA statements. The variables can be either numeric or string and either subscripted or unsubscripted. The READ statement begins reading items from the lowest-numbered DATA statement and proceeds to the next higher-numbered DATA statement as each data list is exhausted. The type (numeric or string) and range (-32768 through + 32767) for integer variables must agree with each item read.

FORMAT

READ list

where:

list represents one or more variables separated by commas.

The READ statement must always be associated with the DATA statements from which it obtains values. An internal data pointer is maintained so that subsequent READ statements read subsequent data items. When a READ statement is executed, the next available data item from a DATA statement is supplied. If a variable in a READ statement is encountered after all data in the DATA statements has been used, an error condition occurs.

EXAMPLES

```
100 READ A,B,C
110 DISPLAY A;B;C;
120 READ A,B,C
130 DISPLAY A;B;C
140 DATA 100,200,300,400,500,600
150 END
```

RUN

```
100 200 300 400 500 600
```

```
10 READ A,A$,B$,B
20 DATA 25, IS, "THE SQUARE OF",5
30 DISPLAY A;A$;B$;B
40 END
```

RUN

```
25 IS THE SQUARE OF 5
```

6.7.3 RESTORE STATEMENT

The RESTORE statement resets the internal data pointer to a specified position. This statement can set the data pointer either to one of the DATA statements in a program or to a record within a file.

6.7.3.1 RESTORE Data Statement Pointer. The RESTORE statement resets the internal data pointer to the lowest-numbered DATA statement in the program or, if you supply a line number, to the DATA statement at the indicated number. The next READ statement executed reads from that point.

FORMATS

RESTORE

RESTORE line__num

If the line number specified does not contain a DATA statement, the pointer is set to the next available DATA statement. If no succeeding DATA statement exists, an error condition occurs when the next READ statement is attempted. The following is an example of the RESTORE statement.

EXAMPLE

```
100 DATA 1,2,3
110 DATA 4,5,6
120 DATA 7,8,9
130 READ A,B,C,D,E,F
140 DISPLAY A;B;C;D;E;F
150 RESTORE
160 READ A,B,C,D,E,F
170 DISPLAY A;B;C;D;E;F
180 RESTORE 110
190 READ A,B,C,D,E,F
200 DISPLAY A;B;C;D;E;F
```

RUN

```
1 2 3 4 5 6
1 2 3 4 5 6
4 5 6 7 8 9
```

6.7.3.2 File RESTORE. The RESTORE statement can move the file position indicator to a specific record in an open file. A file RESTORE is indicated by a unit number preceded by a pound sign (#). If the unit number is not associated with an open file, an error occurs.

FORMATS

RESTORE # unit__number

RESTORE # unit__number, rec__clause

RESTORE # unit__number, key__clause

where:

line__num	specifies the line number of the DATA statement to which the internal data pointer will be reset.
unit__number	is any numeric value or expression greater than 0 and less than 256 that specifies the file.
rec__clause	location of the record; consists of the word REC followed by any legal arithmetic expression.
key__clause	location of the record; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or by a key value.

Sequential File RESTORE. With the sequential file structure, use the RESTORE statement to position the file position indicator to the first record of the file. A subsequent INPUT or ACCEPT statement reads that record.

If the file was opened with the APPEND attribute specified, execution of a RESTORE statement to that file results in an error.

Relative File RESTORE. The RESTORE statement for a relative record file positions the file position indicator to a particular record number in the file. The REC clause indicates the record number. If no REC clause is present, the indicator is positioned at record number 0.

The indicator assigned to unit number 1 is positioned to the eleventh record in the file. The first record is number 0. If the REC clause is omitted, the indicator is positioned to record 0. If the record number specified in the REC clause is greater than the largest record number in the file, the end-of-file (EOF) indicator is set.

EXAMPLE

```
RESTORE #1,REC 10
```

KIF RESTORE. The RESTORE statement for a KIF directs the file position indicator to a specific record based on a key number or key value. Consider the following example:

```
RESTORE #7,KEY #1
```

The KIF assigned to unit number 7 is positioned to the first record in the file according to the sorted order of the primary key.

You can specify a key value to position the file to a record containing that key value as in the following example:

```
RESTORE #9,KEY "ARA"
```

The KIF assigned to unit number 9 is positioned to the first record in the file whose primary key value is equal to or greater than "ARA". Note that if a key number is omitted, the first or primary key is assumed. If more than one record exists that contains the key value specified in the RESTORE statement, the DUP function returns a 1. (See paragraph 8.5.10 for information on the DUP function.)

6.8 PUNCTUATION STATEMENT

The PUNCTUATION statement allows you to modify the following symbols:

- The currency symbol produced by the PRINT or DISPLAY statement with the USING option. Note that the dollar sign (\$) must still be used in output formats (such as IMAGE statements) to indicate that the currency symbol should be printed. The dollar sign is the default currency symbol.
- The decimal point symbol used in input and output data. Numeric constants within the program must use the period (.) as the decimal point. The period is the default decimal point symbol.
- The digit separator produced by the PRINT or DISPLAY statements with the USING option. The comma must be used in formats (such as IMAGE statements) to indicate a digit separator. The comma is the default digit separator symbol.
- The data separator symbol used by the INPUT, PRINT, and DISPLAY statements directed to devices or sequential files. This symbol is produced by executing a PRINT statement with an apostrophe in the output list. The INPUT statement uses this symbol to determine where within a list of data items one item ends and another begins. The comma is the default data separator symbol.

FORMAT

PUNCTUATION str__exp

where:

str__exp represents any valid string expression.

The first four characters in the string expression specify, in order, the characters to be generated for the currency symbol, the decimal point symbol, the digit separator symbol, and the data separator symbol. If the string expression contains more than four symbols, only the first four are used. If the expression contains fewer than four symbols, only those symbols specified are changed; for the omitted characters, the default or previously defined symbols are used. If the data separator symbol is defined to be the same as the decimal point symbol, that symbol is defined as the decimal point symbol when valid. When a BASIC program begins execution, the initial condition is the same as when the following statement is executed:

PUNCTUATION "\$,,"

If French characters are to be used, the following statement is appropriate:

PUNCTUATION "F,:"

This statement defines F as the currency symbol, a comma (,) as the decimal point symbol, a period (.) as the digit separator symbol, and a colon (:) as the data separator. The data separator is modified in this case to prevent it from being confused with the decimal point symbol in input data. I/O statements executed after the PUNCTUATION statement reflect the new notation. The image format used by the PRINT USING option must still adhere to the conventions discussed for the USING option. The following example illustrates the use of the PUNCTUATION statement and its effect on output.

EXAMPLE

```
100 IMAGE PRICE = $$###,###.## QUANTITY = ##
110 PRICE = 123456.789
120 QUANTITY = 4
130 PRINT USING 100 : PRICE, QUANTITY
140 PRINT PRICE'QUANTITY
150 PUNCTUATION "F,:"
160 PRINT USING 100 : PRICE, QUANTITY
170 PRINT PRICE'QUANTITY
180 END
```

RUN

```
PRICE = $123,456.79 QUANTITY = 4
123456.789 , 4
PRICE = F123.456,79 QUANTITY = 4
123456,789 : 4
```


After the PUNCTUATION statement executes, entering multiple data items requires use of the specified data separator to punctuate input from the keyboard. For example, the following statements require that data entered from the keyboard be in the format ABC:DEF:HIJ.

```
100 PUNCTUATION "F,:"  
110 INPUT A$,B$,C$
```

You can use the PUNCTUATION statement to redefine the data separator symbol for sequential and KIF I/O, enabling you to support data with embedded commas, as in the following example:

```
100 OPEN #1: "DS01.MYFILE"  
110 PUNCTUATION "$,:"  
120 A$ = "DOE, JOHN"  
130 B$ = "505 MAIN ST."  
140 PRINT #1: A$'B$'  
150 RESTORE #1  
160 INPUT #1: NAME$,ADDR$  
170 PRINT NAME$  
180 PRINT ADDR$
```

RUN

```
DOE, JOHN  
505 MAIN ST.
```

This example produces the following file record image:

```
DOE, JOHN:505 MAIN ST.:
```

6.9 SHARED FILES

BASIC supports shared files, that is, relative record files or KIFs that can be opened and accessed by several users simultaneously (if you have a multiuser system). To preserve file integrity, BASIC allows you to lock and unlock records in a shared file. Although several users have access to the same file, a locked record provides exclusive (single-user) read and write access. This feature does not provide file security, since any file user can unlock a locked record; however, this feature does ensure that record updates occur one at a time.

The LOCK clause appears in the INPUT or ACCEPT statement and indicates that another user cannot access the record being read until it is unlocked.

FORMATS

INPUT # unit__number, rec__clause, LOCK:variable

INPUT # unit__number, key__clause, LOCK: variable

ACCEPT # unit__number, rec__clause, LOCK: variable

ACCEPT # unit__number, key__clause, LOCK: variable

where:

unit__number is any numeric value or expression greater than 0 and less than 256 that specifies the disk file or output device.

rec__clause is the record number; consists of the word REC followed by any legal arithmetic expression.

key__clause location of the record to be read; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or key value.

variable specifies the data item (or items) to be input.

Do not confuse the LOCK clause used in the INPUT and ACCEPT statements with the LOCK parameter in the SAVE command. The former is used during input to shared files; the latter is used to prevent the program in question from being listed or modified.

The following example illustrates updating a file with and without record locking.

Without Record Locking

1. User A reads a record.
2. User B reads the same record.
3. User A updates his copy of the record and writes the updated record to the file.
4. User B updates his copy of the record and writes the updated record to disk.

Final Result:
User B's update is incorporated in the final record, but user A's update is lost.

With Record Locking

1. User A reads a record and locks it.
2. User B attempts to read the same record, but finding it locked, automatically waits for it to be unlocked.
3. User A updates the record and writes it back to disk, thereby unlocking it.
4. User B reads the record and locks it.
5. User B updates the record and writes it back to disk.

Final Result:
Both user A's and user B's updates are included in the record.

To unlock a record, use the UNLOCK statement.

FORMAT

UNLOCK # unit__number, rec__clause

UNLOCK # unit__number, key__clause

where:

- | | |
|--------------|---|
| unit__number | is any numeric value or expression greater than 0 and less than 256 that specifies the disk file or output device. |
| rec__clause | is a record number; consists of the word REC followed by any legal arithmetic expression. |
| key__clause | location of the record to be read; consists of the word KEY followed optionally by a pound sign (#) and a key number and/or by a key value. |

6.9.1 Relative Record File Example

The following example locks record 10 of a relative record file after it is read. The record remains inaccessible to other users until it is unlocked or rewritten.

```
200 INPUT #2,REC 10,LOCK:DAT(1),DAT(2)
```

The following command unlocks the record from the previous example:

```
500 UNLOCK #2, REC 10
```

6.9.2 KIF Example

To lock a KIF, you insert a LOCK clause in an INPUT or ACCEPT statement following the KEY clause.

The following example locks a record in a KIF. When one user attempts to access a record locked by another, the attempt fails. BASIC reattempts the access several times. If the record remains locked throughout the attempted accesses, BASIC assumes that the record is permanently inaccessible and terminates the requesting program with an error message. For this reason, it is important that the record remain locked for a minimum amount of time.

```
200 INPUT #3,KEY #7 KY$,LOCK:A$,B$,C$
```

The following command unlocks the record from the previous example:

```
600 UNLOCK #3, KEY #7 KY$
```

Control Statements

7.1 INTRODUCTION

Unless directed otherwise, a program executes program statements in sequence from the first through the last statement. Certain statements, described in the following paragraphs, permit you to alter the execution sequence.

7.2 UNCONDITIONAL TRANSFER (GOTO)

The unconditional transfer statement transfers program control to the specified statement. Each time a program encounters an unconditional transfer statement, program execution proceeds from the specified statement.

FORMAT

GOTO line__num

GO TO line__num

where:

line__num is the line number of the statement to which control is to be transferred.

In the following example, the statement in line 600 transfers control to the statement in line 300. At line 300, variable A is incremented by 1 and a new sum, C, is formed. This process continues indefinitely, since the END statement is blocked by line 600 and cannot be executed.

EXAMPLE

```

100 A = 0
200 B = 10
300 A = A + 1
400 C = A + B
500 PRINT C;
600 GO TO 300
700 END

```

RUN

```

11 12 13 14 15 16 17 18 19 . . . . .

```

7.3 COMPUTED TRANSFER (ON-GOTO)

The computed transfer statement provides a multiple switch for altering program control, depending on the value of an expression. The computed transfer statement transfers program control to a designated line as a function of the value of the indicated expression.

FORMAT

ON exp GOTO line__1, line__2, line__3 . . .

where:

exp is any valid arithmetic expression.

line__1 represent the line numbers of the statements to which program control transfers,
line__2 depending on the value of the arithmetic operand.
line__3

A value of 1 (exp = 1) transfers program control to the statement whose line number appears first in the list; the value 2 transfers control to the statement whose line number appears second in the list, and so on. The relationship between the value of the arithmetic expression and the line numbers to which control is transferred is positional. When the value of the arithmetic expression is less than one or greater than the number of line numbers in the statement, an error results. The following example illustrates the computed transfer statement.

EXAMPLE

```
90  X = 0
100 X = X + 1
110 ON X GOTO 120, 140, 120, 160
120 PRINT "AT LINE 120"
130 GO TO 100
140 PRINT "AT LINE 140"
150 GO TO 100
160 PRINT "AT LINE 160"
```

RUN

```
AT LINE 120
AT LINE 140
AT LINE 120
AT LINE 160
```

7.4 CONDITIONAL TRANSFER (IF-THEN-ELSE)

The conditional transfer statement provides greater flexibility in program control. Unlike the GOTO statement, which provides an unconditional transfer of control, the IF-THEN construct passes control only if a specific condition is fulfilled. Otherwise, the normal, sequential execution of program statements continues.

FORMAT

IF condition THEN action__a ELSE action__b

where:

condition is a relational expression.

action__a are either statements or line numbers in the current program.
action__b

The ELSE action__b clause is optional. If the value of the relational expression is true, execution of the program continues with the specified THEN action. If the value is false and the ELSE clause is included, action__b occurs; if the ELSE clause is not included, execution continues with the statement immediately following the IF-THEN statement. Note that the end-of-line terminates IF-THEN-ELSE statements. All statements following THEN are part of the THEN block; similarly, all statements following the ELSE clause, if present, are part of the ELSE block. The entire IF-THEN-ELSE statement must be contained on one line. The following is an example of a conditional transfer.

EXAMPLE

```

100 A$ = "AAA"
110 B$ = "BBB"
120 C$ = "AAABBB"
130 IF C$ = A$ & B$ THEN 160
140 PRINT "AT LINE 140"
150 STOP
160 PRINT "AT LINE 160"
170 END

```

RUN

AT LINE 160

In this example, program control is transferred to line 160 because C\$ equals the concatenation of A\$ and B\$.

The following example is similar to the computed transfer example except that an IF-THEN statement replaces the GOTO statement in line 600. The IF-THEN statement checks the value of the variable C to determine if program control should be shifted to line 300. When C reaches the value 16, the condition is no longer satisfied and the normal sequential execution of program statements continues. The END statement executes, and the program terminates.

```
100 A = 0
200 B = 10
300 A = A + 1
400 C = A + B
500 PRINT C;
600 IF C <= 15 THEN GOTO 300
700 END
```

RUN

```
11 12 13 14 15 16
```

An arithmetic expression can replace the relational expression in the IF statement. The condition is false if the value of the expression is zero and true if the value is nonzero.

In the following example, the variable PASS is used as a flag variable to determine whether the user input is less than 10.

```
10 LET MAX = 10
20 PRINT "ENTER A VALUE";
30 INPUT A
40 IF A >= MAX THEN PASS = 0 ELSE PASS = -1
50 IF PASS THEN PRINT "OK" ELSE PRINT "BAD"
90 END
```


7.5 REPEATED SEQUENCES (FOR-TO-STEP-NEXT)

To execute a set of instructions several consecutive times, you can define a repeating sequence, or loop. Using a FOR statement and its associated NEXT statement, you can repeatedly execute program segments located between pairs of these statements.

FORMATS

```
FOR ind_var = init_val TO lim_val
```

```
FOR ind_var = init_val TO lim_val STEP increment
```

```
NEXT ind_var
```

where:

ind_var can be any unsubscripted numeric variable.

init_val can be any numeric expression.

lim_val can be any numeric expression.

increment can be any numeric expression.

When the FOR statement is executed, the ind_var takes on the value of init_val. Each time the associated NEXT statement is encountered, the increment is added to ind_var. If the amount of the increment is not specified, the value 1 is used. If the increment is positive and the new value of ind_var does not exceed the value of lim_val, execution proceeds from the statement following the FOR. Similarly, if the increment is negative and the value of ind_var is not less than the value of lim_val, execution proceeds from the statement following the FOR.

The BASIC statements executed by the loop are those with line numbers between the FOR statement and the associated NEXT statement. In the following example, the statements between the FOR statement and the NEXT statement are executed until the conditions of the loop defined by the values of A, B, and C are satisfied. The value of A is the initial value of the index, B is the limit value of the loop, and C is the value by which I is incremented each time the loop is executed. In some cases, the loop does not execute. The loop is ignored and the associated NEXT statement is executed in either of the following cases:

- If the increment is positive and the limit is less than the initial value
- If the increment is negative and the limit is greater than the initial value

EXAMPLE

```
100 FOR I = A TO B STEP C
200 PRINT I
300 PRINT I*I
400 NEXT I
500 STOP
```

The following example shows an equivalent set of instructions using the IF statement:

```
100 X1 = B
110 X2 = C
120 I = A
130 IF(I-X1)*SGN(X2)> 0 THEN 180
140 PRINT I
150 PRINT I*I
160 I = I+X2
170 GOTO 130
180 STOP
```

The expressions for starting value, limit, and increment are evaluated when the FOR statement is initially executed. BASIC saves these expressions, separated from user control. The program is then free to modify (within the loop) the value of any variable in these expressions with no effect on the number of times the loop is executed. Loops can be nested within other loops; that is, a FOR and a NEXT statement can reside between another FOR and NEXT statement set. However, the two sets of statements cannot use the same index variable. The following example demonstrates the use of the FOR and NEXT statements in loops.

```

100 FOR I = 1 TO 4
110 FOR J = 1 TO 4
120 READ A,B,C,D
130 PRINT A;B;C;D;
140 NEXT J
145 RESTORE :: PRINT
150 NEXT I
160 DATA 11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44
170 END

```

RUN

```

11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44
11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44
11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44
11 12 13 14 21 22 23 24 31 32 33 34 41 42 43 44

```

7.6 GOSUB AND RETURN STATEMENTS

BASIC programs can have internal subroutines. An internal subroutine is a set of statements accessed by a GOSUB statement. Two statements control the use of internal subroutines: GOSUB and RETURN. GOSUB provides entry to the subroutine, and RETURN exits the subroutine.

FORMAT

```

GOSUB line__num
.
.
.
RETURN

```

The line_num parameter following the GOSUB statement identifies the first line of the subroutine. The RETURN statement causes the program to resume execution at the statement immediately following the GOSUB. The following is an example.

```

430 X = 182
440 Y = 4
450 GOSUB 650
460 PRINT "X DIVIDED BY Y IS "; A
.
.
.
650 A = X / Y
660 RETURN
    
```

After the assignment of values to variables X and Y, the GOSUB statement executes at line 450. The next statement to execute is statement 650. When the RETURN statement executes, execution transfers to the statement following GOSUB (statement 460).

You can nest subroutines so that one subroutine contains a GOSUB to another subroutine (Figure 7-1).

Program	Execution Sequence
10 X = 2	10 X = 2
20 GOSUB 90	20 GOSUB 90
30 PRINT Z	90 IF X <>5 THEN 120
40 X = 5	120 X = 10
50 GOSUB 100	130 GOSUB 190
60 PRINT Z	190 Z = X*2
.	200 RETURN
.	140 Z = Z + 2
.	150 RETURN
90 IF X <>5 THEN 120	30 PRINT Z
100 X = 1	40 X = 5
110 GOTO 190	50 GOSUB 100
120 X = 10	100 X = 1
130 GOSUB 190	110 GOTO 190
140 Z = Z + 2	190 Z = X*2
150 RETURN	200 RETURN
.	60 PRINT Z
.	.
.	.
190 Z = X*2	.
200 RETURN	.

Figure 7-1. Nested Subroutines

In Figure 7-1, execution begins by assigning 2 to the variable X. The GOSUB statement at line 20 transfers control to the subroutine at line 90, which assigns 10 to X. The GOSUB statement at line 130 transfers control to a second (nested) subroutine at line 190, which assigns 20 to Z. Upon execution of the RETURN statement, control returns to statement 140. Z is calculated to a value of 22, and the RETURN statement at line 150 exits the current subroutine. The value of Z is printed at line 30, X is assigned a value of 5, and a GOSUB to line 100 is executed. X is assigned a value of 1, and execution continues at line 190 because of the GOTO statement at line 110. After Z is recalculated, the RETURN statement returns control to the statement following the last executed GOSUB (in this case line 60), where the new value of Z is displayed.

The RETURN statement at line 200 terminates the subroutine that was called at line 130 and began execution at line 190. However, due to the second subroutine call at line 50, coupled with a GOTO statement rather than a GOSUB statement at line 110, the second execution of the RETURN statement at line 200 terminates the subroutine that began at line 100.

7.7 COMPUTED GOSUB STATEMENT

The computed GOSUB statement is similar to the computed GOTO statement and has the following format.

FORMAT

```
ON exp GOSUB line__1, line__2, . . .
```

This statement is identical to the computed GOTO statement except that a RETURN statement returns control to the next statement. The following example uses the ON GOSUB and RETURN statements.

EXAMPLE

```
10 X = 0
20 X = X + 1
30 ON X GOSUB 100,200,300
40 IF X < 3 THEN 20 ELSE 500
100 PRINT "AT SUBROUTINE 100"
110 RETURN
200 PRINT "AT SUBROUTINE 200"
210 RETURN
300 PRINT "AT SUBROUTINE 300"
310 RETURN
500 END
```

```
RUN
```

```
AT SUBROUTINE 100
AT SUBROUTINE 200
AT SUBROUTINE 300
```

In this example, control passes to each succeeding subroutine as the value of X is incremented. Finally, when the value of X reaches 3 and the RETURN statement is executed in the third subroutine, control passes to statement 500 via the ELSE clause at statement 40.

7.8 ON ERROR/RETURN

The ON ERROR statement enables you to gain program control when execution time errors, both recoverable and unrecoverable, occur. The ON ERROR statement specifies the action to be taken when an error occurs. This allows you to “trap” expected errors and process them within the BASIC program without terminating program execution. However, the ON ERROR statement will not trap errors encountered during the execution of a RUN statement within a BASIC program except to verify the presence of the destination program.

NOTE

Error trapping is inoperative under certain conditions. An Out-of-Memory error might indicate insufficient memory to trap the error. Also, certain errors in program structure (such as improper nesting of FOR ... NEXT loops) are detected during prescan of the program and are thus ignored during prescan of the program execution.

The form of the ON ERROR statement is as follows:

FORMAT

ON ERROR line__num

ON ERROR STOP

In this format, line__num is the line number to which control is to be transferred when an error occurs within the program unit. If the line number does not exist, an error occurs when the ON ERROR statement is executed.

The group of statements executed when an error is trapped is referred to as the *error processing routine*, which can be located anywhere within the BASIC program. If you use the STOP clause, a subsequent error causes the default action to occur. That is, the error message is displayed and either the program is halted or the system-provided recovery takes place. When the error routine is entered, the previous error trapping mode is reset and the default error processor is activated. To trap further errors, you must execute an ON ERROR statement prior to exiting the error routine or after the RETURN statement.

The ON ERROR statement is active only in the procedure in which it is executed. If an external subprogram is invoked, the current error processing state is saved and the default error processing state is activated until an ON ERROR statement is executed within the subprogram. When the subprogram is exited, the previous error processing state is reactivated. If an ON ERROR statement is executed in an external subprogram, the error processing state is saved when the subprogram is exited and reinstated on subsequent calls. A subprogram should not specify the same error processing routine as the main program or another subprogram to avoid confusion as to which variables are active in the calling routine.

To resume program execution after an error routine, execute a RETURN statement. The form of the RETURN statement is as follows:

FORMATS

RETURN

RETURN NEXT

RETURN line__num

RETURN PRINT

The RETURN statement specifies that the statement that caused the error is to be executed again. The RETURN NEXT statement specifies that the statement following the one that caused the error is to be executed next.

The RETURN line__num statement indicates the line number of the statement to be executed. Any line number within the current program unit can be used. The RETURN PRINT statement specifies that the error message associated with the current error is to be displayed, and execution is to continue with the statement following the one that caused the error.

By using the ON ERROR statement in conjunction with the ERR intrinsic function (see Section 8) and the RETURN statement, you can control error processing.

EXAMPLE

```

100 ON ERROR 150
110 ACCEPT "ENTER A NUMERIC VALUE:" :A$
120 I = VAL(A$)
130 PRINT I
140 GO TO 110
150 IF ERR <> 2 THEN 180
160 PRINT "VALUE ENTERED MUST BE NUMERIC!"
170 GO TO 190
180 PRINT "UNEXPECTED ERROR ENCOUNTERED—RETRY"
190 RETURN 100

```

If you enter alphabetic characters when the ACCEPT statement is executed at line 110, evaluation of the VAL function in line 120 generates an error condition. The error is trapped because of the execution of the ON ERROR statement at line 100. Control passes to line 150, where the value of the error received is inspected and the appropriate recovery action is taken. The RETURN statement at line 190 returns control to line 100, where the error trap is reset and execution continues.

7.9 END STATEMENT

The END statement indicates that the end of the program has been reached.

FORMAT

END

The last statement of every program in memory is an implicit END statement. When an END statement executes, the program terminates and BASIC enters the command mode.

7.10 STOP STATEMENT

The STOP statement terminates execution of the program at some point prior to the END statement.

FORMAT

STOP

The STOP statement is useful in programs that have more than one logical stopping point. You can also use it to terminate a program during execution when an abnormal condition occurs.

7.11 OTHER METHODS OF TRANSFERRING CONTROL

Several other methods of transferring program control are available to you. Section 8 describes user-defined functions and subprograms. Section 11 describes external assembly language subroutines. Section 12 describes the BASIC sort and Keyed File Package (KFP) subroutines.

Intrinsic Functions

8.1 INTRODUCTION

The intrinsic functions of the TI BASIC programming language are grouped as mathematical functions, string functions, date and time functions, and miscellaneous functions. Table 8-1 lists these functions by category.

Table 8-1. BASIC Intrinsic Functions

Mathematical Functions	String Functions
Absolute Value ABS	Convert ASCII to Decimal ASC
Arctangent ATN	Break BREAK
Cosine COS	Length LEN
Exponential EXP	Numeric NUMERIC
Integer INT	Position POS
Logarithm LOG	Repeat RPT\$
Sign SGN	Match String SPAN
Sine SIN	Uppercase UPRC\$
Square Root SQR	Value VAL
Tangent TAN	Character CHR\$
	Segment SEG\$
	String STR\$
Date and Time Functions	Miscellaneous Functions
Date DAT\$	Random Number RND
Time TIME\$	Randomize RANDOMIZE
	Find Available Space FREESPACE
	Return Number of Characters in Buffer INKEY
	Return Character INKEY\$
	End-of-File EOF
	Verify File Type FTYPE
	Tab TAB
	Err ERR
	Test for Duplicate Keys DUP

When you execute the examples of the functions in the following paragraphs, the system responds with a question mark (?) prompt to allow you to enter a value for the unknown.

8.2 MATHEMATICAL FUNCTIONS

The following paragraphs describe the mathematical functions and their associated forms. Unless otherwise noted, these functions have the following format:

FORMAT

fun__name(arg)

where:

fun__name is a three-letter function name.

arg may be an expression, constant, or variable.

The value of the function applied to the argument replaces the function name in the statement in which it appears. You can use functions instead of variables on the right-hand side of assignment statements, PRINT statements, ON statements, and function definitions.

8.2.1 Absolute Value Function (ABS)

The ABS function returns the absolute value of its argument. It returns a nonnegative argument value unaltered and returns the absolute value of a negative argument.

In the following example, if you entered -1, the system would return a value of 1 as the absolute value of -1.

EXAMPLE

```
10 INPUT X
20 PRINT ABS(X)
30 END
```

EXAMPLE

```
? -1
1
```

8.2.2 Arctangent Function (ATN)

The ATN function returns the angle (in radians) whose tangent is the argument of the function. To obtain the size of the angle in degrees, multiply the number of radians by $180/\pi$.

EXAMPLE

```
10 INPUT X
20 D = ATN(X)*(180/3.14159265)
30 PRINT D
40 END
```

Executing this example produces the following:

```
? 5.9246
80.4194732508
```

8.2.3 Cosine Function (COS)

The COS function returns the cosine of the argument. The argument represents an angle in radians. To convert an angle to radians, multiply the number of degrees by $\pi/180$.

EXAMPLE

```
10 INPUT B
20 PRINT COS(B)
30 END
```

Executing this example produces the following:

```
? 1.25
.315322362395
```

8.2.4 Exponential Function (EXP)

The EXP function returns the value of e (the base of natural logarithms) raised to the power specified in the argument.

EXAMPLE

```
10 INPUT X
20 PRINT EXP(X)
30 END
```

Executing this example produces the following:

```
? 25
72004899337.3
```

8.2.5 Integer Function (INT)

The INT function rounds down to the nearest whole number.

EXAMPLE

```
10 INPUT Z
20 PRINT INT(Z)
30 END
```

Executing this example produces the following:

```
? 3.7
3
```

Note that all computations are performed in floating-point format regardless of the type of variables declared. The result of the expression is converted to the data type of the assigned variable only after the entire expression has been evaluated unless the INT or other functions preempt the normal evaluation. Therefore, in the preceding example, if Z is defined as an integer and the INT function is removed from statement 20, the input of 3.7 is first rounded to 4 and then assigned to the variable Z.

8.2.6 Natural Logarithm Function (LOG)

The LOG function returns the natural logarithm (base e) of the argument.

EXAMPLE

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

Executing this example produces the following:

```
? 5280
8.5716813767
```

8.2.7 Sign Function (SGN)

The SGN function returns the value 0 if the argument X is zero, +1 if X is positive, and -1 if X is negative.

EXAMPLE

```
100 A = SGN(2.3 + 7)
110 B = SGN(ABS(-2.4)-3)
120 C = SGN(0)
130 D = SGN(-0)
150 PRINT "A = ";A;" B = ";B;" C = ";C;" D = ";D
160 END
```

Executing this example produces the following:

```
A = 1 B = -1 C = 0 D = 0
```

The SGN function interprets -0 as zero rather than as negative.

8.2.8 Sine Function (SIN)

The SIN function returns the sine of the argument. The argument represents an angle in radians. To convert an angle from degrees to radians, multiply the number of degrees by $\pi/180$.

EXAMPLE

```
10 INPUT A
20 PRINT SIN(A)
30 END
```

Executing this example produces the following:

```
? 1.25
.948984619356
```

8.2.9 Square Root Function (SQR)

The SQR function returns the value of the square root of the specified argument. The argument can be positive or zero. An error message appears if the argument is negative.

EXAMPLE

```
10 INPUT J
20 PRINT SQR(J)
30 END
```

Executing this example produces the following:

```
? 2
1.41421356237
```

8.2.10 Tangent Function (TAN)

The TAN function returns the tangent of the argument. The argument represents an angle in radians. To convert an angle from degrees to radians, multiply the number of degrees by $\pi/180$.

EXAMPLE

```
10 INPUT N
20 PRINT TAN(N)
30 END
```

Executing this example produces the following:

```
? 0.137
.137863601824
```

8.3 STRING FUNCTIONS

The following paragraphs discuss the various string functions provided by BASIC.

8.3.1 Convert ASCII to Decimal Function (ASC)

The ASC function returns the decimal ASCII value of the first character of the specified string.

FORMAT

ASC(str)

where:

str can be any string expression.

The following is an example:

EXAMPLE

```
10 Y$ = "?"
20 PRINT ASC(Y$)
30 END
```

Executing this example produces the following:

63

The program in this example prints the number 63, which is the decimal representation of the ASCII code for the question mark (?).

EXAMPLE

```
10 LET C = ASC("X")
20 PRINT C
99 END
```

Executing this example produces the following:

88

The program in this example assigns the number representation of the ASCII character X to the variable C. When run, the program prints the number 88.

8.3.2 Break Function (BREAK)

The BREAK function finds the first character in one string that matches any character in a second string.

FORMAT

```
BREAK(str__1,str__2)
```

where:

```
str__1  
str__2
```

are string expressions.

This function compares the first character in string 1 to each character in string 2. If no match occurs, the next character of string 1 is compared to each character in string 2. This process continues until a match occurs or until all characters in string 1 have been checked. The value returned is the number of characters compared in string 1 before a match occurred or, if no match occurs, the total number of characters in string 1.

EXAMPLE

```
100 A$ = "ABCDEFGHJIJ"  
110 B$ = "STRING2"  
120 C$ = "$"  
130 PRINT BREAK (A$, B$)  
140 PRINT BREAK (B$, A$)  
150 PRINT BREAK (A$, C$)  
160 PRINT BREAK (C$, A$)  
170 END
```

Executing this example produces the following:

```
6  
3  
10  
1
```


8.3.3 Length Function (LEN)

The LEN function returns the number of characters in the argument string.

FORMAT

LEN(str)

where:

str is any valid string variable, constant, or expression.

The following is an example:

EXAMPLE

```

100 READ A$, B$, C$
110 PRINT "A$ = ";A$, "LENGTH = ";LEN(A$)
120 PRINT "B$ = ";B$, "LENGTH = ";LEN(B$)
130 PRINT "C$ = ";C$, "LENGTH = ";LEN(C$)
140 DATA ABC, DEFGH, IJKLMNOP
150 END

```

Executing this example produces the following:

A\$ = ABC	LENGTH = 3
B\$ = DEFGH	LENGTH = 5
C\$ = IJKLMNOP	LENGTH = 8

8.3.4 Numeric Function (NUMERIC)

The NUMERIC function determines whether a certain string represents a valid number. It returns - 1 if the string can be passed to the VAL function (paragraph 8.3.9) without error. It returns 0 if applying VAL to the string would cause an error.

FORMAT

NUMERIC(str)

Executing this example produces the following:

EXAMPLE

```
10 A$ = "HELLO":B$ = "37.5"  
20 X = NUMERIC(A$)  
30 PRINT X  
40 Y = NUMERIC(B$)  
50 PRINT Y;VAL(B$)  
60 END
```

Executing this example produces the following:

```
0  
-1 37.5
```

8.3.5 Position Function (POS)

The POS function determines the position of a substring within another string.

FORMAT

POS(str__1, str__2, start)

where:

str__1
str__2 are any two strings.
start is a numeric value.

The value returned indicates the character position within str__1 of the first occurrence of str__2. The search begins at character position start, rounded to an integer. If a substring does not exist as defined, 0 is returned as the value. The following example shows the result of applying the POS function.

EXAMPLE

```
110 PRINT POS("ABCDEFGH ABCD","AB",1)  
120 PRINT POS("ABCDEFGH ABCD","AB",2)  
130 A$ = "STRING1 AND STRING2 ARE ANY TWO STRINGS AND"  
140 I = 1  
150 I = POS(A$,"IN",I)  
160 PRINT I;  
170 IF I <> 0 THEN I = I + 1 ELSE STOP  
180 GOTO 150
```

Executing this example produces the following:

```
1
9
4   16   36   0
```

8.3.6 Repeat Function (RPT\$)

The RPT\$ function provides a string repetition capability. The function uses two variables, a string and a number, and produces a new string equal to the supplied string repeated the specified number of times. The number must be nonnegative and less than 256. A warning condition occurs if the resulting string has a length greater than 255.

FORMAT

RPT\$(str, num__exp)

An example is as follows:

EXAMPLE

```
PRINT RPT$("**8**",3)
*8**8**8*
```

8.3.7 Match String Function (SPAN)

The SPAN function compares characters in one string with characters in a second string until a character in the first string does not match any character in the second string.

FORMAT

SPAN(string__1,string__2)

where:

```
string__1
string__2
```

are string expressions.

This function compares consecutive characters in string__1 to each of the characters in string__2 and stops at the first character in string__1 that does not occur in string__2. The value returned is the position of the last matched character.

EXAMPLE

```
100 S$ = "*****THIS SENTENCE HAS EMBEDDED ASTERISKS"  
110 ASTS = SPAN(S$,"*")  
120 PRINT ASTS;"ASTERISKS PRECEDE THIS SENTENCE"  
130 END
```

Executing this example produces the following:

```
6  ASTERISKS PRECEDE THIS SENTENCE
```

8.3.8 Uppercase Function (UPRC\$)

The UPRC\$ function changes all lowercase letters in the argument string to uppercase letters.

FORMAT

UPRC\$(str)

The value returned is the string variable with all lowercase letters replaced by uppercase letters. Nonalphabetic characters in the argument remain unchanged.

EXAMPLE

```
10 A$ = "ABC"  
20 PRINT UPRC$(A$),  
30 PRINT UPRC$("def"),  
40 PRINT UPRC$("xYz"),  
50 END
```

Executing this example produces the following:

ABC

DEF

XYZ

8.3.9 Value Function (VAL)

The VAL function returns the numeric value of a string expression that represents a valid numeric value.

FORMAT

VAL(str)

The argument str is any valid string expression that is also a valid numeric representation; blanks (leading or trailing) are permitted. This function can convert numbers in string format to numeric values that can be used in arithmetic expressions.

EXAMPLE

```
100 A$ = "12.2"  
110 A = VAL(A$)  
120 PRINT A;2*A  
130 END
```

Executing this example produces the following:

```
12.2 24.4
```

EXAMPLE

```
100 INPUT A$  
110 PRINT VAL(A$)  
120 END
```

Executing this example produces the following:

```
? 5E5  
500000
```

If the string does not represent a number, an error occurs.

8.3.10 Character Function (CHR\$)

The CHR\$ function returns a one-character string that is the ASCII character represented by the argument (rounded to an integer value); the argument must be greater than or equal to 0 and less than or equal to 255.

FORMAT

CHR\$(num__exp)

This function is the inverse of the ASC function. It is often used to generate special control characters in the PRINT statement.

The following example uses the control characters on an 810 printer, advancing the paper to the top-of-form and controlling print size.

EXAMPLE

```
100 OPEN #1:"LP01"  
110 PRINT #1:CHR$(12)&"NORMAL PRINT SIZE"  
120 PRINT #1:CHR$(13)&CHR$(27)&CHR$(55)&"REDUCED PRINT SIZE"  
130 PRINT #1:CHR$(27)&CHR$(54)&"RESTORED PRINT SIZE"  
140 STOP
```

Executing this example produces the following:

```
NORMAL PRINT SIZE  
REDUCED PRINT SIZE  
RESTORED PRINT SIZE
```

The capabilities of other control character sequences are given in the reference manual for the specific device.

8.3.11 Segment Function (SEG\$)

The SEG\$ function extracts the designated segment of the specified string.

FORMAT

SEG\$(str__1, item__1, item__2)

where:

str__1 is the specified string or string-valued expression from which the extraction is to be made.

item__1 is the character position in the specified string at which the desired substring begins.

item__2 is the length of the substring to be extracted.

Item__1 and item__2 can be any arithmetic expression and are rounded to integer values. If item__1 is less than or equal to 0, an error occurs; if item__1 is greater than the length of str__1, the null string is returned; if item__2 is less than 0, an error occurs; if item__2 is equal to 0, the null string is returned; if the sum of item__1 and item__2 is greater than the length of str__1, the remainder of str__1 (starting at the position indicated by item__1) is returned.

EXAMPLE

```
100 A$ = "ABCDEFGHI"
110 B$ = SEG$(A$,2,4)
120 PRINT "B$ = ";B$
130 END
```

Executing this example produces the following:

```
B$ = BCDE
```

8.3.12 String Function (STR\$)

The STR\$ function returns the string representation of the specified numeric argument. This function can be assigned to a string variable or referred to directly in an output statement.

FORMAT

STR\$(arith__exp)

As shown in the following example, STR\$ returns a string value that is the same as if the value had been displayed at the terminal. The first character in the resulting string is either a blank (for positive numbers) or a minus sign (for negative numbers).

EXAMPLE

```
90  FOR I = 1 TO 4
100 INPUT A
110 A$ = STR$(A)
120 PRINT A;LEN(A$);A$
130 PRINT
140 NEXT I
150 END
```

Executing this example produces the following:

```
? 123456789
 123456789      9      123456789

? 1234.567
 1234.567      8      1234.567

? 1.0E2
 100      3      100

? 12345671234.5678
 12345671234.6    13      12345671234.6
```

8.4 DATE AND TIME FUNCTIONS

BASIC provides functions that access the date and time. The following paragraphs describe these functions.

8.4.1 Date Function (DAT\$)

The DAT\$ function provides the calendar date as a string in the following form.

FORMAT

MM/DD/YY

where:

MM indicates the month.

DD indicates the day.

YY indicates the year.

The following example prints the date (January 2, 1980) by using a variable. The DAT\$ function need not be assigned to a string variable.

EXAMPLE

```
10 LET A$ = DAT$
20 PRINT A$
99 END
```

Executing this example might produce the following:

```
01/02/80
```

The following example also prints the date.

```
10 PRINT DAT$
99 END
```

8.4.2 Time Function (TIME\$)

The TIME\$ function indicates the time of day in the form of a string, based on a 24-hour clock. You can either assign the function to a string variable or refer to it directly in an output statement. The form of the output is as follows:

FORMAT

```
HH:MM:SS
```

where:

- HH indicates the hour.
- MM indicates the minute.
- SS indicates the second.

The following example prints the time twice, first using a variable and then using the function directly.

EXAMPLE

```
100 A$ = TIME$
110 PRINT A$, TIME$
120 END
```

Executing this example might produce the following:

11:30:29 11:30:29

8.5 MISCELLANEOUS FUNCTIONS

The miscellaneous functions include the RND, FREESPACE, INKEY, INKEY\$, EOF, FTYPE, ERR, TAB, and DUP functions, plus the RANDOMIZE statement. The following paragraphs describe these functions.

8.5.1 Random Number Function (RND)

The RND function produces pseudorandom numbers, where each number is greater than or equal to 0 and less than 1. The RND function does not require an argument. The same sequence of random numbers is generated for each program that calls the RND function unless the RANDOMIZE statement with no seed value is executed prior to evaluating the RND function. You can specify a seed value by using the RANDOMIZE statement to indicate a new sequence of random numbers. This sequence of numbers is generated each time the RND function is evaluated until another RANDOMIZE statement is executed. The following example shows the result of applying the RND function when using the RANDOMIZE statement with a constant as a seed value.

EXAMPLE

```
100 RANDOMIZE (.354879485439)
110 FOR I= 1 TO 5
120 PRINT RND
130 NEXT I
140 END
```

Executing this example produces the following:

```
.782246790824
.833832091865
.670523262318
.852162175859
.974191040483
```

8.5.2 Randomize Statement (RANDOMIZE)

The RANDOMIZE statement reinitializes the random number generator.

FORMATS

```
RANDOMIZE num__exp
```

```
RANDOMIZE
```

The num__exp is a numeric expression whose value is used as a seed for the random number generator. If num__exp is not provided, the seed value is derived from the real-time clock.

8.5.3 Find Available Space Function (FREESPACE)

The FREESPACE function determines the amount of available space in memory. The number of bytes or characters currently available in memory is returned.

FORMAT

```
FREESPACE(0)
```

In the following example, if the number of bytes available were 500, the statement FREESPACE(0) would set the numeric variable EMPTY equal to 500.

EXAMPLE

```
10 EMPTY = FREESPACE(0)
```

8.5.4 Return Number of Characters in Buffer Function (INKEY)

The INKEY function returns the number of characters in the keyboard buffer. The length of the buffer is operating-system dependent; buffers contain characters in a first-in, first-out (FIFO) queue. If no characters are currently in the queue, 0 is returned.

FORMAT

```
INKEY(0)
```

An example is as follows:

EXAMPLE

```
100 IF INKEY(0) = 0 THEN 100
110 ACCEPT X$
120 DISPLAY X$
```

8.5.5 Return Character Function (INKEY\$)

The INKEY\$ function reads and removes characters from the keyboard buffer. The value returned is a string equal to the next character in the buffer. The INKEY\$ function deletes the character from the input buffer, allowing the next character to be read. The codes returned for INKEY\$ functions differ, depending on the keyboard. Refer to Appendix I for a list of the INKEY\$ function codes and the keys to which they map.

When you execute the following example, the program waits for you to type a character on the screen. That character is then deleted from the input buffer and displayed on the screen.

EXAMPLE

```
10  I$ = INKEY$(0)
20  PRINT I$
30  END
```

8.5.6 End-of-File Function (EOF)

The EOF function determines when the last record of a file has been read. (See Section 6 for a more detailed discussion on reading and writing files.)

FORMAT

EOF(X)

where:

- X is the unit number assigned to the file when the file was opened. X can be any legal arithmetic expression.

This function can return the following values:

- 0 — Indicates the file is not positioned at the end-of-file (that is, an attempt to read the next record will not result in an end-of-file error).
- 1 — Indicates the file is positioned at the end-of-file (that is, an attempt to read the next record will result in an end-of-file error).
- 4 — Indicates the unit number (X) is not being used.

In the following example, 10 records are written to file TEST, and an attempt is made to read 20. However, BASIC sets the end-of-file indicator after reading the tenth record; consequently, the EOF value at statement 50 is 1 when I equals 11. Program execution then transfers to statement 80 with I-1 records read.

EXAMPLE

```
10 OPEN #1: "DS01.TEST",OUTPUT:: A$ = "1234567890"  
20 FOR I = 1 TO 10 :: PRINT #1:A$ :: NEXT I  
30 CLOSE #1  
40 OPEN #1: "DS01.TEST",INPUT  
50 FOR I = 1 TO 20 :: IF EOF(1)= 1 THEN 80  
60 ACCEPT #1:A$ :: PRINT A$  
70 NEXT I  
80 PRINT "END OF FILE HIT AFTER READING";I-1;"RECORDS"  
90 END
```

```
RUN  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890  
1234567890
```

You must check the EOF flag on sequential and relative record files before each read to determine whether any records are left. You should check the EOF flag on a key indexed file (KIF) after each read to determine if the specified record exists. The EOF status of a relative record file may not be reported accurately following a print operation.

8.5.7 Verify File Function (FTYPE)

This function verifies the existence and type of a file or a device. The argument is any valid string variable or string constant that contains the filename or device name.

FORMAT

FTYPE(str)

Table 8-2 lists the values FTYPE returns.

Table 8-2. FTYPE Values

Value	Meaning
-1	Error
0	File not found
1	Busy device
2	Open sequential file
3	Key indexed file
4	Sequential file
5	Device
$x*32 + 6$ (See Note)	Relative record file

Note:

x is the logical record length of the file.

For relative record files, the value returned contains the record size in bits 1 through 10 (which can be accessed by dividing the returned value by 32), and the relative file identifier in the remaining 5 bits (which can be accessed by ANDing the returned value with 31). The following word illustrates how the file length and the file type identifier are returned.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0											0	0	1	1	0

LOGICAL RECORD LENGTH
 IF < = 1023; 0 IF > = 1024

When the record length is not representable in 10 bits (greater than or equal to 1024), a 0 is returned in bits 1 through 10.

EXAMPLES

```

100 I = FTYPE ("PATHNAME")
110 IF I = 0 THEN PRINT "MISSING PAYROLL FILE!"
120 END

100 I = FTYPE ("PATHNAME")
110 IF I <= 0 THEN 140
120 IF (I AND 31) <> 6 THEN 140
130 PRINT "RELATIVE FILE, RECORD LENGTH: "; INT (I/32)
140 END

```

NOTE

If the FTYPE function attempts to access a device, delays in program execution can result. For example, using FTYPE to determine the file type of a terminal can suspend the program until you press Return. Used with a communication port, FTYPE can suspend the program until the device times out. FTYPE does not correctly indicate the presence of a temporary file.

8.5.8 Tab Function (TAB)

The TAB function advances the printhead or cursor to a specific position on the device. Use TAB only in the output list of a PRINT or DISPLAY statement.

FORMAT

TAB(n)

where:

n is the column at which output begins; n can be a numeric constant, variable, or arithmetic expression.

The rounded integer value is used. The printhead is then moved to that position before printing the next character. The value n must be positive and is taken modulus the field length of the device.

If the position designated by the TAB function has already been passed, the function moves to the next line and tabs to the designated position.

EXAMPLE

```
100 PRINT "12345678901234567890"  
110 PRINT TAB(10); 123;  
120 PRINT TAB(12); 1234  
130 PRINT TAB(15);"ABCDEF"  
140 PRINT 123456 :: PRINT TAB(4); 123  
150 END
```

RUN

```
12345678901234567890  
          123  
          1234  
          ABCDEF  
123456  
  123
```

8.5.9 ERR Function

The ERR function returns an integer code that identifies the last warning or error that occurred. The form of this function is as follows:

FORMAT

ERR

If no exception has occurred, ERR returns a zero. When a warning has occurred, ERR returns a negative warning number. When an error has occurred, ERR returns a positive error message number.

EXAMPLE

```

100 ON ERROR 150
110 ACCEPT "ENTER A NUMERIC VALUE: ":I$
120 IF I$ = "" THEN STOP
130 PRINT VAL(I$)
140 GO TO 100
150 IF ERR < 0 THEN PRINT "WARNING "; ELSE PRINT "ERROR ";
160 PRINT "NUMBER: ";ABS(ERR);":OCCURRED"
170 RETURN 100
180 END

```

Executing this example produces the following:

```

ENTER A NUMERIC VALUE: ABC

ERROR NUMBER: 2 :OCCURRED

ENTER A NUMERIC VALUE:

```

Additionally, you can trap I/O errors by using the ON ERROR statement. If error trapping is not in effect at the time of an I/O error, the system returns an error message consisting of the ERR value and the equivalent operating system error code (hexadecimal). The following example shows an operating system error message:

EXAMPLE

```
ERROR #1183 (OS > B7) IN LINE 120
```

where:

ERR value equals 1000 plus the operating system error code (decimal).

This error code indicates that an attempt was made to access a locked record. Refer to your operating system error message manual for details on messages and codes.

8.5.10 Test for Duplicate Keys (DUP)

The DUP function is used to determine the validity of certain KIF operations.

FORMAT

DUP (X)

where:

- x is the unit number assigned to the KIF when the file was opened. X can be any legal arithmetic expression.

The DUP function returns a value of 0 or 1. The function retains its value until the next I/O operation is performed on the argument unit number.

After write operations, the DUP function returns a value of 1 if you attempted to insert a record which would create a duplicate value for a key defined as allowing no duplicates. In that case, the insert operation (PRINT or REPRINT) aborts. If you have defined a key as allowing no duplicates, you should test the value of the DUP function following each insert operation to ensure that the PRINT or REPRINT succeeded.

After the positioning operation (RESTORE with key value specified), the DUP function is also set. It returns a value of 1 if a subsequent read accesses a record with the same key value. The following example shows the application of the DUP function.

EXAMPLE

```
100 INTEGER ALL
110 OPEN #1:"BASIC.KEY",KEYED
120 INPUT "ENTER NAME, AGE, OCCUPATION: ":N$,A$,O$
130 CALL FILL(N$,20)
140 CALL FILL(A$,3)
150 PRINT #1:N$'A$'O$
160 IF DUP(1)=0 THEN 180
170 PRINT "DUPLICATE ENTRY! — INSERT ABORTED."
180 CLOSE #1
190 END
200 ! BLANK FILL PRIMARY & SECONDARY KEYS.
210 SUB INTEGER FILL(K$,LNGTH)
220 IF LEN(K$)< LNGTH THEN K$ = K$ & RPT$(" ",LNGTH-LEN(K$))
230 SUBEND
```

In the preceding example, a KIF was created with two keys. The first (primary) key was created with a length of 20 characters and specified as no duplicates. The second key was created with a length of three characters and specified as allowing duplicates. The DUP function is used to determine whether an attempt is made to insert a record with a duplicate primary key. If such an attempt is made, the DUP function will return a value of 1 and the insert operation will be aborted.

User-Defined Procedures

9.1 INTRODUCTION

BASIC allows you to define single-statement and multiple-statement functions and allows you to use internal and external subprograms. This section describes the definition and use of these procedures within a BASIC program.

9.2 FUNCTION DEFINITION

Function definition involves the define statement (DEF) and the function-end statement (FNEND). A function definition specifies the means of evaluating the function based on the values of the parameters appearing in the parameter list, other variables, or constants. When the function is referenced, the expressions in the argument list of the function reference are evaluated and their values are assigned to the parameters in the parameter list of the function definition. The function is then evaluated, and a value is assigned as the value of the function.

9.2.1 Define Statement (DEF)

The DEF statement can be used for both single- and multiple-statement function definitions. For single-statement definitions, the DEF statement contains the entire function definition; for multiple-statement definitions, the DEF statement defines only the parameters and the local variables used in the body of the function definition. Function definitions are generally placed at the beginning of a program. In the format descriptions that follow, the first DEF statement would be used for single-statement definition and the second for multiple-statement definition.

FORMATS

DEF fun__name

DEF type fun__name = exp

DEF type fun__name (type parm, type parm) = exp

DEF type fun__name type local, type local

DEF type fun__name (type parm, type parm) type local, type local

where:

fun__name is the name of the function.

type clause specifying numeric type: INTEGER, REAL, or DECIMAL.

parm specifies a parameter.

exp is the expression which is evaluated to determine the value of the function.

local specifies a local variable.

The following is an example of a single-statement function definition:

EXAMPLE

```
DEF XYZ(A,B) = A + B + 2
```

In this example, XYZ is the name of the function, A and B are the arguments, and A + B + 2 is the expression that is evaluated to determine the value of the function. Appendix G shows the complete format in the num__fun statement and the str__fun statement.

The following is an example of a multiple-statement function definition:

EXAMPLE

```
DEF XYZ(A,B)
XYZ = A + B + 2
FNEND
```

In this example, XYZ is the name of the function, A and B are the arguments, and $A + B + 2$ is the expression that is evaluated to determine the value of the function. The value of the function is the last value assigned to the function name prior to the execution of FNEND.

The function name can be any legal variable name corresponding to the type of result (numeric or string) returned. If the function is numeric, you can use the optional type specification to declare the type of numeric result (that is, REAL, INTEGER, or DECIMAL). The optional parameter list defines those variables that must appear when the function is referenced. The local variable list, which is optional, defines those variables that do not appear in the parameter list but that are also local to the function definition. The following examples show different forms of the DEF statement and display the distinct characteristics of each.

EXAMPLES

Single-statement function definition:

```
50  DEF AVG(X,Y) = (X + Y)/2
70  DEF REAL PI = 3.1415926535
    .
    .
    .
200 INPUT A
210 INPUT B
220 X = AVG(A,B)/PI
230 PRINT X
```

Multiple-statement function definition:

```
100 DEF SQRT(A) XN,XN1
110 XN = 1
120 XN1 = (XN + A/XN)/2
130 IF XN = XN1 THEN 160
140 XN = XN1
150 GOTO 120
160 SQRT = XN1
170 FNEND
    .
    .
    .
300 INPUT A
310 PRINT SQRT(A)
```

A parameter appearing in the parameter list of a DEF statement is local to the defined function. A local parameter or variable is a variable that is assigned a definition only for the routine in which it is declared. Therefore, two variables can share the same name but have different values depending on the routine being executed.

In the preceding example, XN and XN1 are local variables to the function SQRT. These variables can have one value during execution outside the function and another value during execution of the function. Variables not explicitly defined as local are considered global. A global variable is one that contains only one value, regardless of the routine being executed. The default specification for a variable is global. If you intend for any variable to be local, you must explicitly define it as such at the beginning of the function. A variable can be global to one function and local to another.

9.2.2 Function End Statement (FNEND)

The FNEND statement signifies the end of a multiple-statement function definition. It must be the last statement of the definition. You cannot use FNEND with a single-statement definition.

FORMAT

FNEND

9.2.3 Recursive Functions

You can use functions that have parameters recursively; that is, a function can call itself. The following is an example of a recursive function.

EXAMPLE

```
100 DEF FACT(X)
110 IF X <= 0 THEN FACT = 1 :: GOTO 130
120 FACT = X*FACT(X-1)
130 FNEND
140 PRINT FACT(4)
```

Executing this example produces the following:

24

9.3 BASIC SUBPROGRAMS

BASIC subprograms provide a powerful mechanism for developing programs. If several separate programs use a procedure, it can be coded once as a subprogram and then merged with the programs that use it. By using subprograms, you eliminate the problem of conflicting use of identical variable names, which must be considered if two BASIC programs are merged. This attribute also allows you to partition a large program so that several people can work on it at the same time. For example, more than one application program can use a subprogram that uses values for the year, month, and day to compute the Julian date.

There are two kinds of BASIC subprograms: internal and external. Internal subprograms reside in memory with the main program. External subprograms reside on disk and are loaded into memory when they are called. When an external subprogram completes execution, it is released from memory. Programmers should avoid frequent calls to external subprograms, since accessing the disk causes the program to execute more slowly. You can use external subprograms to advantage with programs that are too large to fit in memory.

9.3.1 Calling Subprograms

The CALL statement transfers control to the named BASIC subprogram. When control passes back to the calling program, execution resumes at the statement immediately following the CALL statement that initiated the subprogram. Arguments are passed to the subprogram in the CALL statement and can be altered by the subprogram. The CALL arguments must agree in number, type, and number of dimensions with the parameters defined in the subprogram's SUB statement. The subprogram may or may not have any arguments.

FORMATS

CALL sub__name

CALL sub__name (parm, parm)

where:

sub__name is the name of the subprogram.

parm specifies a parameter.

The following are examples of the CALL statement:

EXAMPLES

CALL SUBTOTALS

CALL PAYCHECK(SSN(I),PAYRATE(I),DATE,I,TABLE(,))

In the first CALL statement in this example, a subprogram is called without an argument. In the second example, a subprogram is called with several arguments; the first two are numeric array elements, the next two are numeric variables, and the last is a two-dimensional numeric array.

When a subprogram is called, it is said to be active. Before a subprogram is called or after it is exited (by executing either a SUBEND or SUBEXIT statement within it), it is said to be inactive. Subprograms can call other subprograms as long as the subprogram called is inactive.

9.3.2 Subprogram Statements

Four statements define BASIC subprograms: SUB, ESUB, SUBEXIT, and SUBEND.

9.3.2.1 SUB Statement. The SUB statement defines the name of the internal subprogram and the names used within the subprogram to refer to the data (parameters) passed to it.

FORMATS

SUB sub__name

SUB sub__name (parm, parm)

SUB type sub__name (parm, parm)

where:

sub__name is the name of the subprogram.

parm specifies a parameter.

type clause specifying numeric type: INTEGER, REAL, or DECIMAL.

The SUB statement must be the first statement of the subprogram. The parameters appearing in the SUB statement are used as variables within the subprogram. The parameters cannot appear in type statements within the subprogram.

EXAMPLE

```
SUB JULIAN(DAY,MONTH,YEAR,JDATE)
```

In this example, the subprogram is named JULIAN. DAY, MONTH, and YEAR are input parameters to the subprogram. The statements that follow the SUB statement use these variable names and others to calculate the Julian date. The subprogram places the result in the variable JDATE and then returns to the program that referenced it.

Parameters to a subprogram can be arrays. To indicate an array parameter, enter the parameter name followed by a pair of parentheses in the SUB statement. If the array has more than one dimension, place a comma between the parentheses for each additional dimension. In the following example, PART is a simple parameter, VALUE is a one-dimensional array parameter, and VENDOR is a three-dimensional array parameter.

EXAMPLE

```
SUB PRICEOUT(PART,VALUE( ),VENDOR(,,))
```

You can insert a type identifier after SUB and before the subprogram name to specify the default type of variables within the subprogram. For example, SUB INTEGER TIMECARD indicates the beginning of a subprogram within which the untyped variables are INTEGER. This subprogram has no parameters; you can assume that it receives its data from another source, such as a data file. No variables are global to the subprogram.

9.3.2.2 ESUB Statement. The ESUB statement defines the name of the external subprogram and the names used within the subprogram to refer to the data (parameters) passed to it.

FORMATS

```
ESUB esub__name
```

```
ESUB esub__name (parm, parm)
```

```
ESUB type esub__name (parm, parm)
```

where:

esub__name is the name of the external subprogram; it cannot contain more than eight characters.

parm specifies a parameter.

type clause specifying numeric type: INTEGER, REAL, or DECIMAL.

The ESUB statement functions identically to the SUB statement. Refer to the description of the SUB statement for details.

NOTE

An ESUB may not call another ESUB.

9.3.2.3 SUBEXIT Statement. When a SUBEXIT statement is executed, control returns to the calling program. At that time, it is assumed that all operations associated with the subprogram's task have been completed. The parameters are set to the values required by the program that transferred control to the subprogram.

FORMAT

SUBEXIT

9.3.2.4 SUBEND Statement. The SUBEND statement signifies the end of the subprogram text. It performs the same function as a SUBEXIT statement. The SUBEND statement must be the last statement of the subprogram.

FORMAT

SUBEND

9.4 DEFINING TYPES OF PARAMETERS AND LOCAL VARIABLES

Variables in the local variable list of a DEF statement or in the parameter list of a CALL or DEF statement must be defined before you can introduce a type statement. As a result, if the variables are to be of a type other than the default type, you must declare the type in the parameter list or local variable list. Precede each local variable with a type specification, as shown in the following examples.

EXAMPLES

```
DEF STRP(INTEGER X, INTEGER Y,Z)
```

```
SUB SUM__TRIP(TRGNT, INTEGER DEST, DECIMAL(4) MILES)
```

```
DEF INTEGER FIBB(REAL N, REAL ASY, I, SUM) X, INTEGER Z
```

The function STRP is defined to have three arguments. X and Y are of type INTEGER, and Z is the default type. In subprogram SUM__TRIP, the parameter TRGNT is the default type, DEST is INTEGER, and MILES is DECIMAL with a scale factor of 4. Function FIBB has declared the function result type to be INTEGER, but two of its parameters (N and ASY) are specified as REAL. Its local variable Z is INTEGER.

Debug Features

10.1 INTRODUCTION

BASIC provides several features that are useful in locating programming errors. This section describes the explicit statements that support the breakpoint and trace features. Note that you cannot use the debug features on a locked program.

10.2 PROGRAM BREAKS

Program breaks are temporary halts in the execution of a program. You can initiate them either by entering the BRKPNT command or by pressing the break function key. (Appendix I lists the function keys.)

10.2.1 BRKPNT and UNBRKPNT Commands

The BRKPNT command sets one or more program breakpoints, each of which stops the program at a particular line during execution. At command level, the BRKPNT command sets breakpoints in memory at the lines specified in the program. The UNBRKPNT command removes breakpoints. If you specify a nonexistent line number, a warning condition results. You can use a BRKPNT statement with no line numbers specified; the program halts when the BRKPNT statement is executed.

When an external subprogram executes, breakpoints to the main program are removed, except for those set within the program. Similarly, breakpoints to external subprograms, except for those set within the subprogram, are removed when the subprogram returns control to the main program. Therefore, programmers will sometimes need to use explicit BRKPNT commands when debugging programs containing external subprograms.

The following examples illustrate breakpoint commands and their functions.

Example Command	Function
BRKPNT 100,175	Halts execution at the specified lines (issued at command level)
UNBRKPNT	Removes all current breakpoints except those set within a program
UNBRKPNT 100,175	Removes only the breakpoints set with the previous BRKPNT command at lines 100 and 175
10 X = 0 20 X = X + 1 30 IF X < 3 THEN 20 40 BRKPNT	Halts execution when X equals 3

To proceed from a break, press the resume execution function key.

10.2.2 Break Function Key

If you press the break function key while a program is actively running or waiting for input from the keyboard, the system immediately halts the program, displays the line number where execution halted after the message STOPPED IN LINE, and prompts you for command input. At this point, you can enter any command or direct execution statements. To restart the program, press the resume execution function key. If the program is awaiting input, the INPUT statement reexecutes; otherwise, execution continues with the statement at which the break occurred. When you press the break key during active execution, the break occurs at the next statement (not necessarily at the end of the line). Thus, an endless loop contained on one line cannot execute indefinitely. You can display the values by entering the variable name and pressing the calculate function key. Program execution will not continue if you modify the program text while in command mode; an error message will appear. You must enter RUN to restart the program.

10.3 CONTINUING EXECUTION

The resume execution function key resumes program execution after a break caused by either the break function key or a breakpoint. Execution resumes with the line at which the breakpoint occurred. The values of all variables are retained. You can modify these values during a break in execution by executing statements in the command mode. Program execution will not continue if you modify the program text while in command mode; an error message will appear. You must enter RUN to restart the program.

10.4 STEPPING A PROGRAM

Stepping a program consists of executing one statement of a program and following that step with a program break. This procedure is useful for program debugging.

The step function key steps the program. Normally, you should use the BRKPNT command to set breakpoints, stopping the program at specified locations, and then use the step key to continue execution one statement at a time. Program execution does not continue if you modify the program text while in command mode. Note that you cannot step a locked program. You cannot continue execution once you try to step a locked program; an error message will appear. You must enter RUN to restart the program.

10.5 TRACE AND UNTRACE

For debugging, the TRACE command displays the line number of each command as it is executed in a program. The actual program flow is displayed as it is executed so that you can identify endless program loops and other programming problems. You cannot use TRACE in a locked program. You must remove the TRACE command prior to locking a program.

FORMAT

TRACE

Trace output is directed to the screen. Output continues until an UNTRACE command is encountered, either in the program or in command mode. The UNTRACE command terminates the tracing initiated by the TRACE command. To execute the UNTRACE command, first press the break function key while a trace is active and then enter the UNTRACE command. You can also execute the UNTRACE command as a program statement.

FORMAT

UNTRACE

Assembly Language Subroutines

11.1 INTRODUCTION

TI BASIC provides an interface that allows you to start execution of and communication with a routine written in 990/9900 assembly code. This feature allows added flexibility in designing a BASIC program. Typical uses of the assembly language subroutine interface include optimizing compute-bound sections of a program and implementing special functions and features not supplied in BASIC.

A system external to the BASIC system creates and formats an assembly language routine. The LIBRARY and CALL statements in a BASIC program then reference the routine. This section describes how to create and install an assembly language routine, how to start execution of the routine and return control to the BASIC program from the routine, and how to communicate data to the routine in the various data formats.

11.2 CREATING AN ASSEMBLY LANGUAGE SUBROUTINE

If you wish to write an assembly language routine for use with a BASIC program, you should have enough knowledge of 990/9900 assembly code to develop, test, and debug the routine before interfacing it to the BASIC program. Refer to the Assembly Language Programmer's Guide for your operating system.

The following restrictions apply to the source code for the assembly language routine:

- You cannot use the first word of the routine.
- The second word must contain the address to which control should be transferred when the BASIC program begins execution of the routine.
- The last instruction to be executed by the routine must be an RTWP instruction.
- When the RTWP instruction is executed to return control to a BASIC program, workspace registers 13, 14, and 15 must contain the same values as they did when the routine was entered.

The following general restrictions apply while you are creating an assembly language subroutine:

- The routine must be small enough to fit into your BASIC user's workspace along with the program that will call it.
- The routine should not attempt input or output with an active BASIC terminal or with a device that a program has already opened for input or output.

11.3 INSTALLING AN ASSEMBLY LANGUAGE SUBROUTINE

When the source code of an assembly language subroutine is ready for interfacing to a BASIC program, you must link the source code in a manner that will allow BASIC to access it properly. First, you should use the Execute Macro Assembler (XMA) command to assemble the source code, sending the object code to a sequential file in standard ASCII object format. Then, you should convert the object code to image format by specifying IMAGE format in the link edit control file and entering the Execute Link Edit (XLE) command.

The name assigned to the linked object that the Link Editor creates is the name by which the BASIC program refers to the assembly language subroutine. Paragraph 11.9 provides an example of the procedure you must follow to create, assemble, link, and then call an assembly language subroutine from a BASIC program.

11.4 LIBRARY STATEMENT

The LIBRARY statement identifies an assembly language subroutine to a BASIC program. The name of each assembly language subroutine to be accessed by a program must appear in a LIBRARY statement before the program can execute the subroutine. The LIBRARY statement must physically precede the CALL statement in the BASIC program that executes the routine. The LIBRARY statement cannot occur in an external subprogram.

FORMAT

LIBRARY "pathname"

LIBRARY "*"pathname"

An asterisk preceding the assembly language subroutine pathname indicates that the subroutine is to be preloaded. The assembly language subroutine is loaded into memory when the RUN statement is executed and remains in memory throughout execution of the BASIC program. This allows the subroutine to retain its internal variable values during multiple calls by the BASIC program. If the asterisk does not appear in the LIBRARY statement, the assembly language subroutine is loaded into memory each time a CALL statement specifying that subroutine is executed.

11.5 CALL STATEMENT

The CALL statement transfers control from the BASIC program to the assembly language subroutine. It also specifies the information to be communicated to the subroutine. The information is passed in a parameter list that specifies the names of variables containing the information. A maximum of 15 parameters can be passed. Any variable can be passed as a parameter to the subroutine; however, attempting to pass a constant value or function as a parameter causes an error.

FORMAT

```
CALL "pathname"(parameter__1,...,parameter__n)
```

11.6 PARAMETER INFORMATION BLOCK

To access the parameters specified in the CALL statement, an assembly language subroutine must calculate a pointer to the parameter information block. Figure 11-1 illustrates a parameter information block.

The location of the parameter information block is calculated by subtracting 36 from the workspace pointer of the assembly language routine. To do so, use the Store Workspace Pointer (STWP) and Add Immediate (AI) assembly language statements as follows:

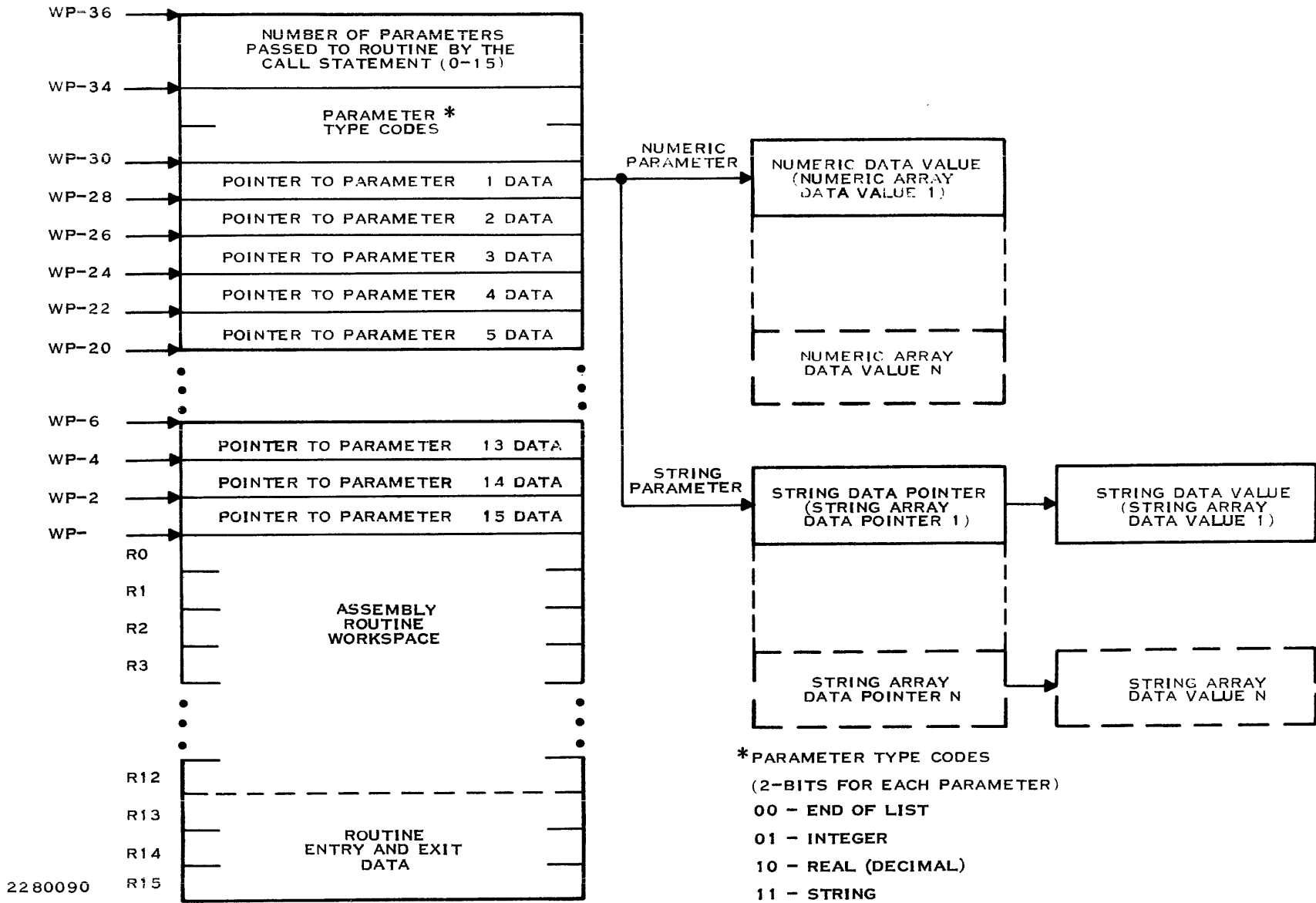
```
STWP   R1
AI     R1,-36
```

If the preceding instruction sequence is executed upon entry to the assembly language subroutine, register 1 (R1) contains a pointer to the parameter information block.

The first word of the block contains the number of parameters occurring in the CALL statement that transferred control to the routine. The second and third words of the block contain information about the types of parameters listed in the CALL statement. A two-bit code represents the type of each parameter occurring in the CALL statement, reading from left to right. The following codes are used:

Code	Meaning
00	End of parameter typing list
01	Integer number
10	Real number (or decimal number)
11	String value

Words 4 through 18 of the parameter information block contain pointers that access the parameters listed in the CALL statement. Each one-word pointer is matched with a corresponding parameter, proceeding from left to right through the parameter list in the CALL statement.



22 80090

Figure 11-1. Parameter Information Block

11.7 ACCESSING PARAMETER DATA VALUES

The pointers in the parameter information block access the data values that allow the BASIC program and assembly language subroutine to communicate. The following sections discuss how to use these pointers to access the data and describe the representation of the data that is passed.

11.7.1 Integer Data Format

The parameter pointer to an integer data value points to a 16-bit word containing a 16-bit, two's-complement integer value. Figure 11-2 shows the representation of integer data. Any 16-bit value in the range -32768 to +32767 is a legal integer value in BASIC. Figure 11-1 shows the parameter pointer trail used to locate an integer parameter.

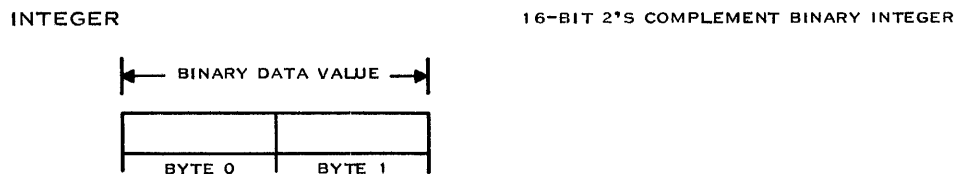


Figure 11-2. Integer Data Format

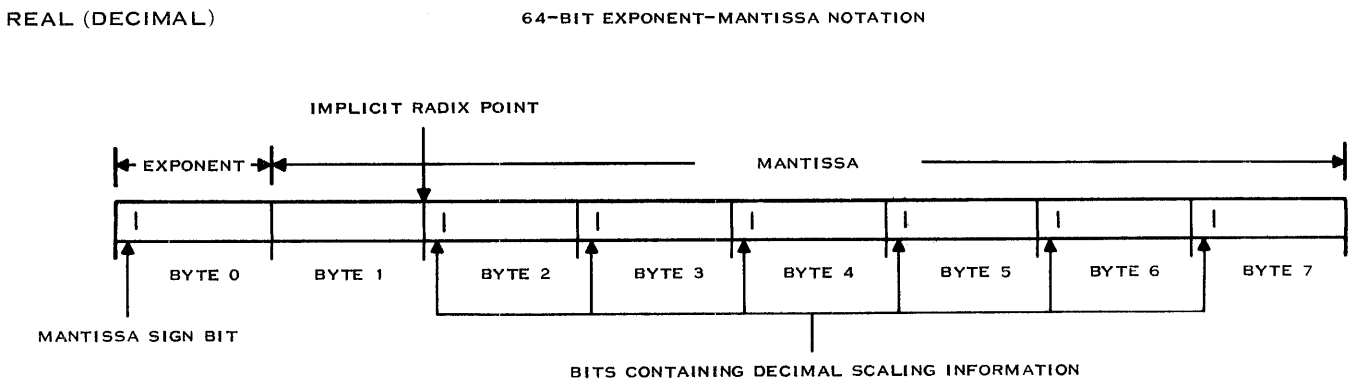


Figure 11-3. Real Data Format

Bit 0 of byte 0 is the sign bit of the mantissa. If bit 0 is set to 0, the mantissa is positive; if it is set to 1, the mantissa is negative. If the sign bit is set to 1, the two's complement of the first word of the four-word value is used to compute the absolute value of the number represented. Bits 1 through 7 of byte 0 contain the radix 100 exponent of the value biased by 64. This means that 64 must be subtracted from the value represented by these bits to obtain the true value of the exponent. (For example, an exponent of 0 is represented as 64, an exponent of -3 is represented as 61, and an exponent of + 2 is represented as 66.) Since the exponent is based on radix 100, the value of the exponent obtained by removing the bias must be multiplied by 2 to obtain the base 10 exponent. (For example, 100 squared is equal to 10 raised to the fourth power.)

Byte 1 of the value contains the first nonzero digit or digits of the radix 100 mantissa. The allowable values for this byte are 1 through 99. Bytes 2 through 7 of the value contain the remainder of the mantissa digits. The allowable values for these bytes are 0 through 99. The implicit radix point is located before the first byte of the mantissa (between bytes 0 and 1 of the 8-byte value). Figure 11-4 shows the parameter pointer trail used to locate a real parameter.

The first byte of a real value can contain any legal eight-bit value in the range 0 through 255; however, spurious results can occur if the allowable bounds of the mantissa bytes are exceeded. The only case in which the first byte of the mantissa can equal 0 is when the data value is 0. A data value of 0 is represented by the first two bytes of the 8-byte value being set to 0.

The following shows some real numbers and their corresponding hexadecimal internal representations:

Real Number	Hexadecimal Numbers			
1	4001	0000	0000	0000
-1	BFFF	0000	0000	0000
103	4101	0300	0000	0000
-103	BEFF	0300	0000	0000
10.3	400A	1E00	0000	0000

11.7.3 Decimal Data Format

Decimal data is represented in the same way as real data except that the highest-order bits (the zero bits) of bytes 2 through 7 have special meanings. These bits store the scaling information that allows a decimal value to retain the precision defined by the BASIC DECIMAL type statement. The example below shows the representation of decimal data.

Bit 0 of byte 2 distinguishes between real data and decimal data. If the bit is set to 0, the 8-byte value is a real value; if the bit is set to 1, the 8-byte value is a decimal value. The 0 bits of bytes 3 through 7 represent the actual scaling factor of the decimal value in the following manner. These bits represent a five-bit value from 1 to 31 when assembled in reverse order; that is, bit 0 of byte 7 becomes bit 0 of the value, bit 0 of byte 6 becomes bit 1 of the value, and so on.

This 5-bit value represents the negative of the scaling factor of the decimal value biased by 16. For example, a scaling factor of 0 is represented by 16, a scaling factor of - 2 is represented by 14, and so on. This scaling factor represents the radix 10 precision that should be maintained in the decimal value. For example, a scaling factor of - 2 means that a precision of two digits to the left of the decimal place should be maintained, while a scaling factor of + 3 means that zeros should be maintained in the first three digits to the right of the decimal place.

With the exception of the decimal scaling information, decimal values are accessed and manipulated in the same way as real values, as shown in the following example:

Real Numbers	Hexadecimal Numbers				Comment
12345.67890	4201	97AD	C35A	0080	Decimal scaling 3
12345.67900	4201	97AD	43D9	0080	Decimal scaling 5
12345.67890	4201	172D	4359	0000	No decimal scaling

11.7.4 String Data Format

Double indirection accesses string parameter data. The parameter pointer points to the string data pointer, which points to a variable-length string data value. Figure 11-4 shows the representation of string data.

The first byte of the data value contains the length of the actual string data in characters (bytes). The first byte can have any value from 0 through 255; 0 indicates that the string is null. BASIC interprets the actual characters in the string data value as standard 8-bit ASCII codes. Figure 11-1 shows the parameter pointer trail used to locate a string parameter data value.

String data passed to an assembly language subroutine as a parameter must be initialized to an adequate length for application by the BASIC program that calls the subroutine. This is because you cannot change the length byte of a string data value (the first byte of the string parameter data value). If an assembly language routine changes the length of the string value, internal pointers may be destroyed, requiring you to reload the BASIC system to repair the damage.

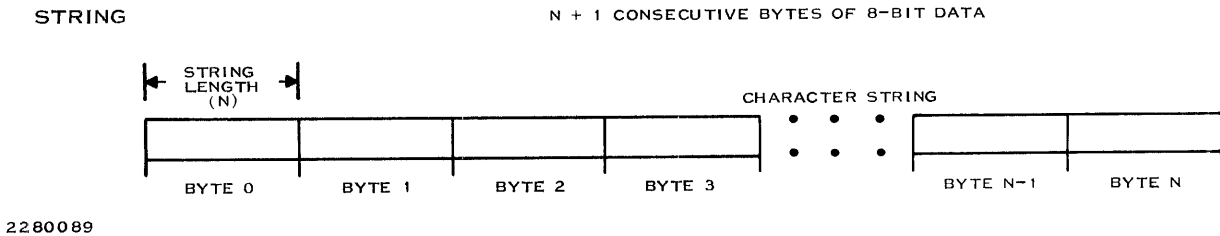


Figure 11-4. String Data Format

11.8 ACCESSING ARRAYS IN AN ASSEMBLY LANGUAGE SUBROUTINE

An array cannot be explicitly passed from a BASIC program to an assembly language subroutine. However, you can gain access to the elements of the array. The techniques required to gain such access also enable an assembly language subroutine to access buffers that it cannot statically allocate. In this case, the BASIC program that calls the subroutine can dimension a numeric array to the required size and allow the routine to use the memory space allocated to the designated array as the required buffer.

When accessing arrays and calculating array element addresses, BASIC has a default index base of 0 for all arrays unless you use the OPTION BASE 1 statement. Consequently, if an array A is dimensioned to have maximum index values of (M1,M2, . . .,Mn), the total number of elements in each dimension is M1 + 1,M2 + 1, . . .,Mn + 1. If you use an OPTION BASE statement to set the array index base to 1 in the program in which array A appears, the total number of elements in each dimension equals M1,M2, . . .,Mn.

BASIC stores numeric array data sequentially. When BASIC accesses multidimensional arrays, the memory location of an element in the array is determined for adjacent array indexes as follows: the index on the left remains fixed, while the index on the right increments from the lowest value to the highest value of its range.

Use the general array formula in the following example to calculate parameter pointer offsets to array elements. Let A be an array with dimensions (M1,M2, . . .,Mn). Let the desired element of A have the array index values of (I1,I2, . . .,In). If the array base of the BASIC program in which A appears is 0 (the default base in BASIC), evaluation of the formula results in the offset. Adding the offset to the parameter pointer (the initial array element pointer) results in a pointer that points to the memory location where the data information is stored.

In the preceding example, D equals 2 when the array is an integer string array and 8 when the array is a real or decimal array. When a BASIC program contains an OPTION BASE statement that sets the array index base to 1, use the following general array formula. This formula compensates for the absence of index 0 elements.

11.8.1 Accessing Numeric Arrays

To access a numeric array, the BASIC program that calls the assembly language subroutine must dimension the array either implicitly or explicitly; the program must then pass the first element of the array to the subroutine as a parameter. To access an element of the array, use the general array formula to compute the offset. The offset is added to the first parameter pointer to set the pointer to the desired array element. For example, if an integer array ARR is dimensioned to (7,8,9), the following is the offset added to the initial parameter pointer to access element ARR(2,3,5):

$$2*(2*(8 + 1)*(9 + 1) + 3*(9 + 1) + 5) = 430$$

If the BASIC program in which array ARR appears includes an OPTION BASE statement that sets the array index base to 1, the offset to be added to access the same element is as follows:

$$2*(2*8*9 + 3*9 + 5) = 352$$

When an assembly language subroutine needs to use BASIC array storage as a buffer, you should dimension a single-dimension integer array to the same number of elements as 16-bit memory words needed for the buffer. The initial parameter pointer passed to the subroutine then contains a pointer to a buffer of the required length.

11.8.2 Accessing String Arrays

To access a string array, the BASIC program that calls the assembly language subroutine must dimension the array either implicitly or explicitly; the program then passes the element of the array to the subroutine. However, in accessing an element string array, the pointers to the string data values (not the string data values themselves) are stored sequentially in memory. Thus, after the program calculates the offset to the desired string array element pointer, the result is added to the pointer parameter; this parameter points to the string data pointer, which in turn points to the string data value. Figure 11-1 illustrates the method by which an array element is accessed.

To calculate the offset to the parameter pointer, use the general array formula, with D equal to 2. For example, if a string array ARR is dimensioned to (2,4,8) and the required element is (2,1,7), the offset added to the pointer is as follows:

$$2*(2*(4+1)*(8+1) + 1*(8+1) + 7) = 212$$

If the BASIC program in which array ARR\$ appears contains an OPTION BASE statement that sets the array index base to 1, the offset added to access the pointer to the same element is as follows:

$$2*(2*4*8 + 1*8 + 7) = 158$$

11.9 ASSEMBLY LANGUAGE SUBROUTINE EXAMPLE

Use the following procedure to prepare and execute any assembly language subroutine to be called by BASIC:

1. Using a Text Editor, create the source file of the assembly language subroutine (Figure 11-5).
2. Assemble the source file into a standard ASCII object file. (Figure 11-6 shows the sample routine object listing, and Figure 11-7 shows the cross-reference listing.)
3. Use the Link Editor to format the object file into an image format object file. (Figure 11-8 shows the sample routine for the link control file.)

The file is now ready to be called by BASIC as an assembly language subroutine.

11.9.1 Step 1 — Creating the Assembly Language Source Code

Use the Text Editor to create the source file of the assembly language subroutine. Figure 11-5 shows a sample routine source code.

```

IDT  'SIGDIG'
TITL 'SIGNIFICANT DIGIT TRUNCATOR'
*****
*   THIS IS A TI 990 BASIC CALLABLE ASSEMBLY LANGUAGE
*   SUBROUTINE.  IT TRUNCATES A REAL VALUE TO A
*   SPECIFIED NUMBER OF SIGNIFICANT DIGITS.  AN ASCII
*   STRING IS RETURNED TO BASIC TO INDICATE WHETHER
*   THE TRUNCATION RESULTED IN A LOSS OF ANY NON-ZERO
*   DIGITS.
*
*   PARAMETERS:
*
*       PARM 1 (REAL DATA TYPE) - ON INPUT CONTAINS
*       THE VALUE WHICH SHOULD BE TRUNCATED BY THIS
*       ROUTINE.  ON OUTPUT CONTAINS THE TRUNCATED
*       VALUE.
*
*       PARM 2 (INTEGER DATA TYPE) - ON INPUT
*       CONTAINS THE NUMBER OF SIGNIFICANT DIGITS
*       TO WHICH THE REAL VALUE IN PARM 1 SHOULD
*       BE TRUNCATED.  NOT CHANGED BY THIS ROUTINE
*
*       PARM 3 (STRING DATA TYPE) - ON INPUT
*       CONTAINS A STRING OF 41 CHARACTERS.  ON
*       OUTPUT CONTAINS A 41 CHARACTER STRING
*       DESCRIBING WHETHER THE VALUE IN PARM 1
*       LOST ANY NON-ZERO DIGITS BECAUSE OF THE
*       TRUNCATION.
*
*       PARM 4 (INTEGER DATA TYPE) - ON INPUT IS
*       INITIALIZED TO A -1.  ON OUTPUT CONTAINS ONE
*       OF THE FOLLOWING COMPLETION CODES:
*
*           -1 PARAMETER TYPE OR NUMBER MISMATCH
*            0 SUCCESSFUL COMPLETION
*            1 STRING PARM OF INCORRECT LENGTH
*            2 ERROR IN NUMBER OF DIGITS REQUESTED
*
*****
PAGE
DATA 0,SIG010          ROUTINE  ENTRY DATA
-----*
OFFSET EQU  -36          OFFSET TO PARM INFO BLOCK
NPARMS EQU   4          NUMBER OF PARAMETERS
PTYPES EQU  >9D00      PARAMETER TYPE WORD
-----*
STRERR EQU  1          STRING PARAMETER ERROR
DIGERR EQU  2          DIGIT REQUEST ERROR
-----*
LB00  BYTE >00
LB0A  BYTE >0A
LW000A EQU LB00
LW000D DATA >000D
-----*
STRLEN DATA MSGLEN*256  STRING PARAMETER LENGTH * 256
VALLEN DATA 8          LENGTH OF REAL PARAMETER
EXPMSK DATA >FF80      EXPONENT MASK FOR REAL VALUE
EXPZRO DATA >0040      EXPONENT BIAS VALUE
-----*
NTRMSG TEXT 'NO TRUNCATION OF NON-ZERO DIGITS'
TRUMSG TEXT 'TRUNCATION CAUSED LOSS OF NON-ZERO DIGITS'
MSGLEN EQU  $-TRUMSG

```

Figure 11-5. Sample Routine Source Code (Sheet 1 of 3)

```

PAGE
SIG010 EQU $
        STWP R0          GET PTR TO PARM INFO BLOCK
        AI   R0,OFFSET
*-----*
        MOV  *R0+,R1     CHECK FOR CORRECT NUMBER AND
        AI   R1,-NPARMS TYPE OF PARMS, ERROR RETURN
        JNE  SIG020     IF INCORRECT
        MOV  *R0+,R1
        CI   R1,PTYPES
        JEQ  SIG030
SIG020 EQU $
        B   @RETURN
*-----*
SIG030 EQU $
        INCT R0          GET POINTERS TO PARAMETERS
        MOV  *R0+,R1     LOCATIONS:
        MOV  *R0+,R2     R1 - PTR TO REAL EXPONENT
        MOV  *R0+,R3     R2 - PTR TO SIG DIGIT COUNT
        MOV  *R0,R4      R4 - PTR TO EQUAL FLAG
        MOV  R1,R5      R5 - PTR TO REAL MANTISSA
        INC  R5
*-----*
        CLR  *R4        CLEAR ERROR FLAG
*-----*
        MOV  *R3,R3
        MOVB *R3+,R6     CHECK STRING PARAMETER FOR
        CB   R6,@STRLEN CORRECT LENGTH; RETURN ERROR
        JEQ  SIG040     IF NOT; IF CORRECT, INITIALIZE
        LI   R0,STRERR  WITH MESSAGE INDICATING NO
        B   @ERROR      TRUNCATION OF NON-ZERO DIGITS
SIG040 EQU $
        MOV  @STRLEN,R6
        SWPB R6
        LI   R7,NTRMSG
        MOV  R3,R8
SIG050 EQU $
        MOVB *R7+,*R8+
        DEC  R6
        JGT  SIG050
*-----*
        MOV  *R2,R6     CHECK NUMBER OF SIGNIFICANT
        JGT  SIG070     DIGITS REQUESTED; RETURN ERROR
SIG060 EQU $          WHEN NEGATIVE, ZERO OR GREATER
        LI   R0,DIGERR THAN SUPPORTED PRECISION
        B   @ERROR
SIG070 EQU $
        CB   R6,@LW000D
        JGT  SIG060
*-----*
        MOV  R1,R7     INITIALIZE R7 TO POINT AT END
        A   @VALLEN,R7 OF REAL MANTISSA
*-----*
        MOVB *R1,R8     STRIP MANTISSA SIGN BIT
        SWPB R8        CHECK SIGN OF EXPONENT; SKIP
        SZCB @EXPMSK,R8 SIGNIFICANT ZERO COUNT WHEN
        MOV  @EXPZRO,R9 EXPONENT IS ZERO OR GREATER
        S   R8,R9
        JEQ  SIG080
        JLT  SIG080
        DEC  R9
        SLA  R9,1      SUBTRACT NUMBER OF SIGNIFICANT
                        ZEROS FROM DIGITS DESIRED; GET

```

Figure 11-5. Sample Routine Source Code (Sheet 2 of 3)


```

S      R9,R6          MORE DIGITS WHEN POSITIVE
JGT   SIG090        RETURN ZERO WHEN NOT POSITIVE
MOVB  @LB00,*R1
JMP   SIG120
*-----*
SIG080 EQU  $
CB    *R5,@LB0A     COMPENSATE FOR NON-SIGNIFICANT
JLT   SIG100        DIGIT IN FIRST MANTISSA BYTE
*-----*
SIG090 EQU  $
DEC   R6            FIND MANTISSA BYTE CONTAINING
JEQ   SIG110        LAST SIGNIFICANT DIGIT; GO TO
SIG100 EQU  $        APPROPRIATE TRUNCATE ROUTINE
INC   R5
DEC   R6
JEQ   SIG120
JMP   SIG090
*-----*
SIG110 EQU  $
CLR   R8            CHECK FOR TRUNCATION OF A
CLR   R9            NON-ZERO DIGIT INSIDE BYTE;
MOVB  *R5,R9        IN NON-ZERO TRUNCATION OCCURS,
SWPB  R9            CHANGE STRING PARAMETER
MOV   @LW000A,R10
DIV   R10,R8
MOV   R9,R6
MPY   R10,R8
SWPB  R9
MOVB  R9,*R5+
MOV   R6,R6
JNE   SIG140
*-----*
SIG120 EQU  $
MOV   R5,R8        CHECK FOR TRUNCATION OF A
SIG130 EQU  $        NON-ZERO BYTE; IF NON-ZERO
MOVB  *R8,*R8+     TRUNCATION OCCURS, CHANGE
JNE   SIG140        STRING PARAMETER
C     R7,R8
JGT   SIG130
JMP   RETURN
*-----*
SIG140 EQU  $
MOV   @STRLEN,R6   MODIFY STRING PARAMETER TO
SWPB  R6            INDICATE TRUNCATION OF
LI    R9,TRMSG     NON-ZERO DIGITS
SIG150 EQU  $
MOVB  *R9+,*R3+
DEC   R6
JGT   SIG150
*-----*
SIG160 EQU  $
MOVB  @LB00,*R5+   CLEAR NON-SIGNIFICANT BYTES IN
C     R7,R5        MANTISSA
JGT   SIG160
*-----*
RETURN EQU  $
RTWP
*-----*
ERROR  EQU  $
MOV   R0,*R4       SET ERROR FLAG TO INDICATE
JMP   RETURN
END

```

Figure 11-5. Sample Routine Source Code (Sheet 3 of 3)

11.9.2 Assembling the Source File

Use the XMA command to assemble the source code into object format. Figure 11-6 shows the object listing you would get if you assembled the sample source code (Figure 11-5). If you select the cross-reference option, as is shown on the first page of the listing, you get the listing of cross references shown in Figure 11-7.

11.9.3 Step 3 — Linking the Assembly Language Subroutine

After you assemble the source code into standard ASCII format, use the Text Editor to create a link control file and then use the XLE command to format the object file into IMAGE format.

Figure 11-8 provides an example of a link control file:

```
FORMAT IMAGE,REPLACE
PHASE 0,EXAMPLE
INCLUDE .OBJ.EXAMPLE
END
```

The XLE command prompts for a linked output access name; the name you assign is the name by which the BASIC program will reference the assembly language subroutine. In our example, assume you assign the name ASMBLY.

11.9.4 Step 4 — Calling the Subroutine from a BASIC Program

Once you have assembled and linked the assembly language subroutine, you can call it from a BASIC program, as illustrated in paragraph 12.2.

```

SDSMAC 3.4.0 81.117 16:10:44 THURSDAY, JAN 20, 1983.
ACCESS NAMES TABLE PAGE 0001

SOURCE ACCESS NAME= VOL.SIGDIGS
OBJECT ACCESS NAME= VOL.SIGDIGO
LISTING ACCESS NAME= VOL.SIGDIGL
ERROR ACCESS NAME=
OPTIONS= XREF,TUNLST
MACRO LIBRARY PATHNAME=
    
```

```

SIGDIG SDSMAC 3.4.0 81.117 16:10:44 THURSDAY, JAN 20, 1983.
PAGE 0002
    
```

```

0001 IDT ^SIGDIG^
0003 *****
0004 * THIS IS A TI 990 BASIC CALLABLE ASSEMBLY LANGUAGE *
0005 * SUBROUTINE. IT TRUNCATES A REAL VALUE TO A *
0006 * SPECIFIED NUMBER OF SIGNIFICANT DIGITS. AN ASCII *
0007 * STRING IS RETURNED TO BASIC TO INDICATE WHETHER *
0008 * THE TRUNCATION RESULTED IN A LOSS OF ANY NON-ZERO *
0009 * DIGITS. *
0010 * *
0011 * PARAMETERS: *
0012 * *
0013 * PARM 1 (REAL DATA TYPE) - ON INPUT CONTAINS *
0014 * THE VALUE WHICH SHOULD BE TRUNCATED BY THIS *
0015 * ROUTINE. ON OUTPUT CONTAINS THE TRUNCATED *
0016 * VALUE. *
0017 * *
0018 * PARM 2 (INTEGER DATA TYPE) - ON INPUT *
0019 * CONTAINS THE NUMBER OF SIGNIFICANT DIGITS *
0020 * TO WHICH THE REAL VALUE IN PARM 1 SHOULD *
0021 * BE TRUNCATED. NOT CHANGED BY THIS ROUTINE *
0022 * *
0023 * PARM 3 (STRING DATA TYPE) - ON INPUT *
0024 * CONTAINS A STRING OF 41 CHARACTERS. ON *
0025 * OUTPUT CONTAINS A 41 CHARACTER STRING *
0026 * DESCRIBING WHETHER THE VALUE IN PARM 1 *
0027 * LOST ANY NON-ZERO DIGITS BECAUSE OF THE *
0028 * TRUNCATION. *
0029 * *
0030 * PARM 4 (INTEGER DATA TYPE) - ON INPUT IS *
0031 * INITIALIZED TO A -1. ON OUTPUT CONTAINS ONE *
0032 * OF THE FOLLOWING COMPLETION CODES: *
0033 * *
0034 * -1 PARAMETER TYPE OR NUMBER MISMATCH *
0035 * 0 SUCCESSFUL COMPLETION *
0036 * 1 STRING PARM OF INCORRECT LENGTH *
0037 * 2 ERROR IN NUMBER OF DIGITS REQUESTED *
0038 * *
0039 *****
    
```

Figure 11-6. Sample Routine Object Listing (Sheet 1 of 5)

```

SIGDIG      SDSMAC 3.4.0 81.117   16:10:44 THURSDAY, JAN 20, 1983.
SIGNIFICANT DIGIT TRUNCATOR                                     PAGE 0003

0041 0000 0000          DATA 0,SIG010          ROUTINE ENTRY DATA
      0002 0062^
0042          *-----*
0043      FFDC  OFFSET EQU  -36          OFFSET TO PARM INFO BLOCK
0044      0004  NPARMS EQU   4          NUMBER OF PARAMETERS
0045      9D00  PTYPES EQU  >9D00      PARAMETER TYPE WORD
0046          *-----*
0047      0001  STRERR EQU   1          STRING PARAMETER ERROR
0048      0002  DIGERR EQU   2          DIGIT REQUEST ERROR
0049          *-----*
0050 0004   00  LB00   BYTE >00
0051 0005   0A  LBOA   BYTE >0A
0052          0004^ LW000A EQU  LB00
0053 0006 000D  LW000D DATA >000D
0054          *-----*
0055 0008 2900  STRLEN DATA MSGLEN*256      STRING PARAMETER LENGTH * 256
0056 000A 0008  VALLEN DATA 8              LENGTH OF REAL PARAMETER
0057 000C FF80  EXPMSK DATA >FF80          EXPONENT MASK FOR REAL VALUE
0058 000E 0040  EXPZRO DATA >0040          EXPONENT BIAS VALUE
0059          *-----*
0060 0010   4E  NTRMSG TEXT `NO TRUNCATION OF NON-ZERO DIGITS`
0061 0039   54  TRUMSG TEXT `TRUNCATION CAUSED LOSS OF NON-ZERO DIGITS`
0062          0029  MSGLEN EQU  $-TRUMSG

```

Figure 11-6. Sample Routine Object Listing (Sheet 2 of 5)

SIGDIG SDSMAC 3.4.0 81.117 16:10:44 THURSDAY, JAN 20, 1983. PAGE 0004
 SIGNIFICANT DIGIT TRUNCATOR

```

0064      0062^ SIG010 EQU $
0065 0062 02A0      STWP R0          GET PTR TO PARM INFO BLOCK
0066 0064 0220      AI   R0,OFFSET
          0066 FFDC
0067
0068 0068 C070      *-----*
0069 006A 0221      MOV  *R0+,R1          CHECK FOR CORRECT NUMBER AND
          006C FFFC      AI   R1,--NPARMS      TYPE OF PARMS, ERROR RETURN
0070 006E 1604      JNE  SIG020          IF INCORRECT
0071 0070 C070      MOV  *R0+,R1
0072 0072 0281      CI   R1,PTYPES
          0074 9D00
0073 0076 1302      JEQ  SIG030
0074 0078 0078^ SIG020 EQU $
0075 0078 0460      B    @RETURN
          007A 0138^
0076
0077      007C^ SIG030 EQU $
0078 007C 05C0      INCT R0          GET POINTERS TO PARAMETERS
0079 007E C070      MOV  *R0+,R1          LOCATIONS:
0080 0080 C0B0      MOV  *R0+,R2          R1 - PTR TO REAL EXPONENT
0081 0082 C0F0      MOV  *R0+,R3          R2 - PTR TO SIG DIGIT COUNT
0082 0084 C110      MOV  *R0+,R4          R4 - PTR TO EQUAL FLAG
0083 0086 C141      MOV  R1,R5          R5 - PTR TO REAL MANTISSA
0084 0088 0585      INC  R5
0085
0086 008A 04D4      *-----*
          CLR  *R4          CLEAR ERROR FLAG
0087
0088 008C C0D3      *-----*
0089 008E D1B3      MOV  *R3,R3
0090 0090 9806      MOVB *R3+,R6          CHECK STRING PARAMETER FOR
          0092 0008^      CB  R6,@STRLEN      CORRECT LENGTH; RETURN ERROR
0091 0094 1304      JEQ  SIG040          IF NOT; IF CORRECT, INITIALIZE
0092 0096 0200      LI   R0,STRERR      WITH MESSAGE INDICATING NO
          0098 0001
0093 009A 0460      B    @ERROR          TRUNCATION OF NON-ZERO DIGITS
          009C 013A^
0094 009E 009E^ SIG040 EQU $
0095 009E C1A0      MOV  @STRLEN,R6
          00A0 0008^
0096 00A2 06C6      SWPB R6
0097 00A4 0207      LI   R7,NTRMSG
          00A6 0010^
0098 00A8 C203      MOV  R3,R8
0099 00AA 00AA^ SIG050 EQU $
0100 00AA DE37      MOVB *R7+,*R8+
0101 00AC 0606      DEC  R6
0102 00AE 15FD      JGT  SIG050
0103
0104 00B0 C192      *-----*
          MOV  *R2,R6          CHECK NUMBER OF SIGNIFICANT
0105 00B2 1504      JGT  SIG070          DIGITS REQUESTED; RETURN ERROR
0106 00B4 00B4^ SIG060 EQU $          WHEN NEGATIVE, ZERO OR GREATER
0107 00B4 0200      LI   R0,DIGERR      THAN SUPPORTED PRECISION
          00B6 0002
0108 00B8 0460      B    @ERROR
          00BA 013A^
0109 00BC 00BC^ SIG070 EQU $
0110 00BC 9806      CB  R6,@LW000D
          00BE 0006^
0111 00C0 15F9      JGT  SIG060
    
```

Figure 11-6. Sample Routine Object Listing (Sheet 3 of 5)

SIGDIG SDSMAC 3.4.0 81.117 16:10:44 THURSDAY, JAN 20, 1983. PAGE 0005
 SIGNIFICANT DIGIT TRUNCATOR

```

0112 *-----*
0113 00C2 C1C1      MOV  R1,R7      INITIALIZE R7 TO POINT AT END
0114 00C4 A1E0      A    @VALLEN,R7  OF REAL MANTISSA
      00C6 000A^
0115 *-----*
0116 00C8 D211      MOVB *R1,R8      STRIP MANTISSA SIGN BIT
0117 00CA 06C8      SWPB R8          CHECK SIGN OF EXPONENT; SKIP
0118 00CC 5220      SZCB @EXPMSK,R8 SIGNIFICANT ZERO COUNT WHEN
      00CE 000C^
0119 00D0 C260      MOV  @EXPZRO,R9  EXPONENT IS ZERO OR GREATER
      00D2 000E^
0120 00D4 6248      S    R8,R9
0121 00D6 1308      JEQ  SIG080
0122 00D8 1107      JLT  SIG080
0123 00DA 0609      DEC  R9          SUBTRACT NUMBER OF SIGNIFICANT
0124 00DC 0A19      SLA  R9,1        ZEROS FROM DIGITS DESIRED; GET
0125 00DE 6189      S    R9,R6        MORE DIGITS WHEN POSITIVE
0126 00E0 1506      JGT  SIG090      RETURN ZERO WHEN NOT POSITIVE
0127 00E2 D460      MOVB @LB00,*R1
      00E4 0004^
0128 00E6 1016      JMP  SIG120
0129 *-----*
0130          00E8^  SIG080 EQU  $
0131 00E8 9815      CB  *R5,@LB0A   COMPENSATE FOR NON-SIGNIFICANT
      00EA 0005^
0132 00EC 1102      JLT  SIG100     DIGIT IN FIRST MANTISSA BYTE
0133 *-----*
0134          00EE^  SIG090 EQU  $
0135 00EE 0606      DEC  R6          FIND MANTISSA BYTE CONTAINING
0136 00F0 1304      JEQ  SIG110     LAST SIGNIFICANT DIGIT; GO TO
0137 00F2^  SIG100 EQU  $  APPROPRIATE TRUNCATE ROUTINE
0138 00F2 0585      INC  R5
0139 00F4 0606      DEC  R6
0140 00F6 130E      JEQ  SIG120
0141 00F8 10FA      JMP  SIG090
0142 *-----*
0143          00FA^  SIG110 EQU  $
0144 00FA 04C8      CLR  R8          CHECK FOR TRUNCATION OF A
0145 00FC 04C9      CLR  R9          NON-ZERO DIGIT INSIDE BYTE;
0146 00FE D255      MOVB *R5,R9     IN NON-ZERO TRUNCATION OCCURS,
0147 0100 06C9      SWPB R9         CHANGE STRING PARAMETER
0148 0102 C2A0      MOV  @LW000A,R10
      0104 0004^
0149 0106 3E0A      DIV  R10,R8
0150 0108 C189      MOV  R9,R6
0151 010A 3A0A      MPY  R10,R8
0152 010C 06C9      SWPB R9
0153 010E DD49      MOVB R9,*R5+
0154 0110 C186      MOV  R6,R6
0155 0112 1606      JNE  SIG140
0156 *-----*
0157          0114^  SIG120 EQU  $
0158 0114 C205      MOV  R5,R8      CHECK FOR TRUNCATION OF A
0159          0116^  SIG130 EQU  $  NON-ZERO BYTE; IF NON-ZERO
0160 0116 DE18      MOVB *R8,*R8+  TRUNCATION OCCURS, CHANGE
0161 0118 1603      JNE  SIG140     STRING PARAMETER
0162 011A 8207      C    R7,R8
0163 011C 15FC      JGT  SIG130
0164 011E 100C      JMP  RETURN
0165 *-----*
    
```

Figure 11-6. Sample Routine Object Listing (Sheet 4 of 5)

SIGDIG SDSMAC 3.4.0 81.117 16:10:44 THURSDAY, JAN 20, 1983. PAGE 0006
 SIGNIFICANT DIGIT TRUNCATOR

```

0166      0120^ SIG140 EQU $
0167 0120 C1A0      MOV @STRLEN,R6      MODIFY STRING PARAMETER TO
      0122 0008^
0168 0124 06C6      SWPB R6          INDICATE TRUNCATION OF
0169 0126 0209      LI R9,TRUMSG      NON-ZERO DIGITS
      0128 0039^
0170      012A^ SIG150 EQU $
0171 012A DCF9      MOVB *R9+,*R3+
0172 012C 0606      DEC R6
0173 012E 15FD      JGT SIG150
0174      *-----*
0175      0130^ SIG160 EQU $
0176 0130 DD60      MOVB @LB00,*R5+      CLEAR NON-SIGNIFICANT BYTES IN
      0132 0004^
0177 0134 8147      C R7,R5          MANTISSA
0178 0136 15FC      JGT SIG160
0179      *-----*
0180      0138^ RETURN EQU $
0181 0138 0380      RTWP
0182      *-----*
0183      013A^ ERROR EQU $
0184 013A C500      MOV R0,*R4          SET ERROR FLAG TO INDICATE
0185 013C 10FD      JMP RETURN
0186      END
NO ERRORS,      NO WARNINGS
    
```

Figure 11-6. Sample Routine Object Listing (Sheet 5 of 5)

SIGDIG SDSMAC 3.4.0 81.117 16:10:44 THURSDAY, JAN 20, 1983. PAGE 0007

SIGDIG LABEL	VALUE	DEFN	REFERENCES
\$	013E		0062 0064 0074 0077 0094 0099 0106 0109 0130 0134 0137 0143 0157 0159 0166 0170 0175 0180 0183
DIGERR	0002	0048	0107
ERROR	013A	0183	0093 0108
EXPMSK	000C	0057	0118
EXPZRO	000E	0058	0119
LB00	0004	0050	0052 0127 0176
LB0A	0005	0051	0131
LW000A	0004	0052	0148
LW000D	0006	0053	0110
MSGLEN	0029	0062	0055
NPARMS	0004	0044	0069
NTRMSG	0010	0060	0097
OFFSET	FFDC	0043	0066
PTYPES	9D00	0045	0072
R0	0000		0065 0066 0068 0071 0078 0079 0080 0081 0082 0092 0107 0184
R1	0001		0068 0069 0071 0072 0079 0083 0113 0116 0127
R10	000A		0148 0149 0151
R2	0002		0080 0104
R3	0003		0081 0088 0088 0089 0098 0171
R4	0004		0082 0086 0184
R5	0005		0083 0084 0131 0138 0146 0153 0158 0176 0177
R6	0006		0089 0090 0095 0096 0101 0104 0110 0125 0135 0139 0150 0154 0154 0167 0168 0172
R7	0007		0097 0100 0113 0114 0162 0177
R8	0008		0098 0100 0116 0117 0118 0120 0144 0149 0151 0158 0160 0160 0162
R9	0009		0119 0120 0123 0124 0125 0145 0146 0147 0150 0152 0153 0169 0171
RETURN	0138	0180	0075 0164 0185
SIG010	0062	0064	0041
SIG020	0078	0074	0070
SIG030	007C	0077	0073
SIG040	009E	0094	0091
SIG050	00AA	0099	0102
SIG060	00B4	0106	0111
SIG070	00BC	0109	0105
SIG080	00E8	0130	0121 0122
SIG090	00EE	0134	0126 0141
SIG100	00F2	0137	0132
SIG110	00FA	0143	0136
SIG120	0114	0157	0128 0140
SIG130	0116	0159	0163
SIG140	0120	0166	0155 0161
SIG150	012A	0170	0173
SIG160	0130	0175	0178
STRERR	0001	0047	0092
STRLEN	0008	0055	0090 0095 0167
TRUMSG	0039	0061	0062 0169
VALLEN	000A	0056	0114

Figure 11-7. Sample Routine Object Listing Cross-Reference


```
FORMAT IMAGE,REPLACE  
TASK EXAMPLE  
INCLUDE VOL. SIGDIGO  
END
```

Figure 11-8. Link Control File for Sample Routine

BASIC Subroutine Library

12.1 INTRODUCTION

This section describes the library of subroutines available with BASIC. The general types are as follows:

- Sort subroutine
- Keyed File Package (KFP) subroutines

You receive the BASIC subroutines in image format on the disk. The subroutines are ready to use with the BASIC program; you should not attempt to modify or alter them.

If the subroutines are copied to a volume other than the system disk, the call statement must use that disk's volume name or the drive name.

The subroutine examples in this section assume that the subroutines are contained on a disk in drive DS01 and can be called using the ".XXXXXX" syntax (where XXXXXX is the name of the file containing the utility).

12.2 USING THE SUBROUTINES

A BASIC program must use the LIBRARY and CALL statements to access the subroutines. In a BASIC program, the LIBRARY statement must precede the CALL statement and must occur in the main program, as shown below:

```

10 LIBRARY ".SORT" {or LIBRARY "* .SORT"}
.. .....
.. .....

60 CALL ".SORT"(DUMY,TYPE,INNAM$,OUTNAM$,WK1$,WK2$,SIZ,RFT$,KFT$,STAT)
.. .....

```

For further information about using the LIBRARY and CALL statements and creating user-defined subroutines, see Section 11.

12.2.1 Subroutine Arguments

Communication between your BASIC program and the subroutine occurs through the argument list. Each subroutine requires that the CALL statement supply a list of arguments in a specified order. The argument list is contained in parentheses after the pathname of a CALL statement. The CALL statement must supply all of the required arguments; otherwise, an error condition results. Also, the arguments must be of the appropriate type (either integer or string) and must be of sufficient length to accept the returned values.

You can name subroutine arguments any name valid for the required data type. If you use arguments that specify arrays, you must specify the dimensions of each array in the appropriate statement at the beginning of the program. Arguments that specify character strings must have names that terminate with a dollar sign. Subroutines cannot be called using literal numeric or string data. For example:

Invalid calling sequence:

```
100 CALL “.SORT”(1,0,“.INPUT”,“.OUTPUT”,“DS01”,“MYDISK”,20, 1:UP,2:UP,0)
```

Valid calling sequence:

```
100 INTEGER ALL
110 DUMY = 0
120 TYPE = 0
130 INNAM$ = “.INPUT”
140 OUTNAM$ = “.OUTPUT”
150 WK1$ = “DS01”
160 WK2$ = “MYDISK”
170 SIZ = 20
180 RFT$ = “$, $”
190 KFT$ = “1:UP,2:UP”
200 STAT = 0
210 CALL “.SORT”(DUMY,TYPE,INNAM$,OUTNAM$,WK1$,WK2$,SIZ,RFT$,KFT$,STAT)
```

In the example of the valid calling sequence, all of the arguments are set to initial values. Failure to initialize an argument, even an output argument, results in an error in the CALL statement.

12.2.2 Subroutine Error Codes

The system indicates an error in subroutine execution in either of two ways:

- The interpreter issues a program line error message for the CALL statement, halting execution.
- The subroutine returns an error code to the program without interrupting execution.

In the first case, the interpreter issues an error message such as the following:

```
ERROR #74, in 120
```

This type of error message appears when any of the following occurs:

- A subroutine CALL statement supplies the incorrect number of arguments.
- The data types of the arguments do not match the data types required by the subroutine.
- The argument is not properly initialized as specified in the subroutine description.

After the message appears on the screen, execution stops and a period is displayed in the lower left-hand corner of the screen. This indicates that the system is in the command mode. You can use the calculate (F1) function to examine the current argument values and determine the source of the error. You can then correct and rerun the program.

In the second method of indicating a subroutine error, the subroutine loads an error code into one of the arguments. This argument is called the *subroutine error status* (STAT). This type of error does not directly halt program execution; however, the subroutine failure may cause an error within the program.

See Figure 12-3 for an example of a BASIC program that checks for error conditions. Appendix D lists and describes the error messages and codes.

12.3 SORT SUBROUTINES

BASIC provides a subroutine that sorts relative record files. This subroutine previously was available only on SBC 990 BASIC.

The Sort subroutine generates the following types of output:

- Sorted record output
- Integer index and sort key output
- Integer index output
- Real index output
- Integer virtual array index output
- Real virtual array index output

The first output type produces a relative record file containing records in sorted order. The other output types produce files or virtual arrays containing pointers that index the input records in the order produced by the sort. The pointers can be integer numeric values, real numeric values, integer virtual arrays, or real virtual arrays. Depending on the output type, this subroutine sorts files with the following maximum sizes:

File Type	Type Size
Sorted record output	System limit
Integer index and sort key	32,767 records
Integer index output	32,767 records
Real index output	6,533,599 records
Integer virtual array output	32,767 records
Real virtual array output	32,767 records

The limits are typically smaller, due to the requirement that files be contiguous and do not span disks.

This subroutine allows a maximum of 100 data fields in an input record. The data fields can be of any type (that is, integer, real, or string). Of these data fields, up to 10 may be designated as sort keys.

NOTE

In the LIBRARY and CALL statements in the following examples, SORT is a synonym for the pathname of the SORT subroutine.

To execute this subroutine, use the following CALL statement:

```
100 CALL "SORT"(DUMY,TYPE,INNAM$,OUTNAM$,WK1$,WK2$,SIZ,RFT$,KFT$,STAT)
```

Argument details are summarized below:

Argument	Type	I/O	Use
DUMY	Integer	Input	Reserved for future use
TYPE	Integer	Input	Output type code:* 0 = Sorted relative records 1 = Integer index and sort key 2 = Integer index 3 = Real index 4 = Integer virtual array 5 = Real virtual array
INNAM\$	String	Input	Pathname of relative record file to be sorted
OUTNAM\$	String	Input	Pathname of output file*
WK1\$	String	Input	First temporary work volume name*
WK2\$	String	Input	Second temporary work volume name*
SIZ	Integer	Input	Maximum size (bytes) of combined keys*
RFT\$	String	Input	Record-format template*
KFT\$	String	Input	Key-field template*
STAT	Integer	Output	Subroutine error status*

* Argument notes and special considerations:

- TYPE This argument specifies the type of output produced by the subroutine. The following values produce the indicated results:
- 0 — The subroutine generates an output file containing the input records in sorted order.
 - 1 — The subroutine generates an output file of integers that index the input file according to the sort specifications. Also, each output record contains the composite key for the indexed record. (The composite sort key consists of the combination of the key fields in the specified hierarchical order.)

- 2 — The subroutine generates an output file of integers that index the input file according to the sort specifications.
- 3 — The subroutine generates an output file with records containing real numbers that index the input file according to the sort specifications.
- 4 — The subroutine generates an integer virtual array that indexes the input file.
- 5 — The subroutine generates a real virtual array that indexes the input file.

OUTNAM\$ This argument specifies the pathname of the file to which the sorted output will be written. The subroutine automatically creates a relative record file with the pathname specified by this argument. If the file already exists, the subroutine aborts and issues an error message. Therefore, if a file with this pathname already exists, delete it before running this subroutine. When sorted record output is specified (TYPE = 0), the input and output files should be on different disks for maximum efficiency. Any file specified by this pathname cannot be open when the Sort subroutine is executed. Also, any virtual array specified by this pathname cannot be assigned when the subroutine is executed.

WK1\$ and WK2\$ These two arguments specify the volumes (or disks) that the Sort subroutine uses to contain intermediate files used in executing the sort. The volumes specified by these arguments should be on different disks for maximum sorting speed. The same volume can be designated for each; however, the time required to complete a sort may be significantly increased. During the sort, the subroutine creates two temporary files, T1\$\$X and T2\$\$X. You should not have any other files with these names, since they are deleted upon completion of the sort.

SIZ This argument specifies the length (in bytes) of the composite key. The composite key is the concatenation of all of the designated key fields. Each integer field has a length of two bytes; each real field has a length of eight bytes; and each string field has a length of one byte per character plus one length byte. Thus, if a sample composite key contained four integer fields, two real fields, and one string field (the string field having a maximum length of 12 characters), then the SIZ parameter would equal the following:

$$\text{SIZ} = (4 \times 2) + (2 \times 8) + ((12 \times 1) + 1) = 37 \text{ bytes}$$

RFT\$ This argument defines the record-format template that specifies the data type descriptors for each field in the record. The descriptors are as follows:

I = Integer

R = Real

\$ = String

A descriptor may be repeated several times by preceding it with an n^* , where n is the number of fields of that type. For example, 4^*R indicates four real fields. Descriptors must be separated by commas. Up to 100 fields can be described.

KFT\$ This argument defines the key-field template that specifies on which fields the file is to be sorted. Starting in order of precedence, the sort key fields are specified by field number. Each must be followed by “:UP” for ascending order or “:DN” for descending order. For example, “2:UP” sorts on the second field of the record in ascending order. The field order specification must be separated by commas. You can specify up to 10 sort key fields.

STAT This argument returns the subroutine error code or completion status. This subroutine prefixes all I/O error codes with 1, 2, or 3 to denote an error for an input, output, or temporary file, as follows (xxx is an error code listed in Appendix D):

1xxx — I/O error on the input file

2xxx — I/O error on the output file

3xxx — I/O error on a temporary file

For example, the error code 2086 indicates that an invalid pathname was specified for the output file.

12.3.1 Sort Example

Figure 12-1 is a program that illustrates the use of the sort subroutine in a BASIC program. It may also serve as a general sort utility program. It prompts you for argument values and then sorts the file as specified.

```

100 INTEGER ALL
110 LIBRARY "SORT"
120 DISPLAY ERASE ALL :: DISPLAY "SORT EXAMPLE"
130 ACCEPT "                OPERATION TYPE: ":T
140 ACCEPT "                INPUT FILE: ":INNAM$
150 ACCEPT "                OUTPUT FILE: ":OUTNAM$
160 ACCEPT "WORK FILE 1 VOLUME OR DEVICE NAME: ":WK1$
170 ACCEPT "WORK FILE 2 VOLUME OR DEVICE NAME: ":WK2$
180 ACCEPT "                MAXIMUM FIELD SIZE: ":SIZ
190 ACCEPT "                KEY(S):UP/DN: ":KEY$
200 ACCEPT "                DESCRIPTORS: ":DESC$
210 !
220 DEL OUTNAM$
230 DISPLAY :: DISPLAY "START TIME:      "&TIME$
240 !
250 DUMY=0 :: ER=0
260 CALL "SORT" (DUMY,T,INNAM$,OUTNAM$,WK1$,WK2$,SIZ,DESC$,KEY$,ER)
270 !
280 DISPLAY "END TIME:      "&TIME$
290 DISPLAY "ERROR RETURNED: "&STR$(ER)

```

Figure 12-1. Sort Subroutine Program

12.3.1.1 Sorted Record Output. The following example illustrates using the sort subroutine driver program to produce sorted relative record output. The input file DS01.INFILE contains records with two alphabetic strings per record. You enter the following parameters into the sort utility program (Figure 12-1) to begin the sorting operation:

Prompt	Response
	OPERATION TYPE: 0
	INPUT FILE: DS01.INFILE
	OUTPUT FILE: DS01.OUTFILE
WORK FILE 1 VOLUME OR DEVICE NAME:	DS01
WORK FILE 2 VOLUME OR DEVICE NAME:	DS02
MAXIMUM FIELD SIZE:	20
KEY(S):UP/DN:	1:UP,2:UP
DESCRIPTORS:	,\$,\$

Note that both fields in the input record are used as sorting keys. Additional fields in the record would be associated with that record in a tag-along fashion. The file is then sorted as follows:

Rec #	DS01.INFILE			Rec #	DS01.OUTFILE	
0	ZEIGLER	JANE		0	ABBOTT	MARY
1	ABBOTT	MARY		1	BEST	ANN
2	SMITH	ALICE		2	ELLSWORTH	JOE
3	SMITH	JAMES	sort	3	GOTWORTH	JUNE
4	JONES	ARTHUR	→	4	JONES	ARTHUR
5	MARTIN	ALEX		5	LYNN	GEORGIA
6	GOTWORTH	JUNE		6	MARTIN	ALEX
7	ELLSWORTH	JOE		7	SMITH	ALICE
8	BEST	ANN		8	SMITH	JAMES
9	LYNN	GEORGIA		9	ZANES	CARL
10	ZANES	CARL		10	ZEIGLER	JANE

12.3.1.2 Integer Output. To produce integer index output, you can enter the following responses into the sort subroutine driver program (Figure 12-1):

Prompt	Response
OPERATION TYPE:	2
INPUT FILE:	DS01.INFILE
OUTPUT FILE:	DS01.OUTFILE
WORK FILE 1 VOLUME OR DEVICE NAME:	DS01
WORK FILE 2 VOLUME OR DEVICE NAME:	DS02
MAXIMUM FIELD SIZE:	20
KEY(S):UP/DN:	1:UP,2:UP
DESCRIPTORS:	,\$

The sorting process proceeds as follows:

Rec #	DS01.INFILE			Rec #	DS01.OUTFILE	
0	ZEIGLER	JANE		0		10
1	ABBOTT	MARY		1		0
2	SMITH	ALICE		2		7
3	SMITH	JAMES	sort	3		8
4	JONES	ARTHUR	→	4		4
5	MARTIN	ALEX		5		6
6	GOTWORTH	JUNE		6		3
7	ELLSWORTH	JOE		7		2
8	BEST	ANN		8		1
9	LYNN	GEORGIA		9		5
10	ZANES	CARL		10		9

The output file for this sort operation contains 11 records, each containing an integer that indexes the record number in sorted order. Note that the index begins with record 0, not record 1. In the above example, record 0 (ZEIGLER JANE) of the input file becomes the last record of the ordered output, and record 1 (ABBOTT MARY) of the input file becomes record 0 of the ordered output.

12.4 KEYED FILE PACKAGE

BASIC supports a Keyed File Package (KFP) that allows you to access relative record files by a key value. All keyed file operations must be performed by calling assembly language subroutines that are supplied with the system from within a user program.

NOTE

Because the KFP buffer is destroyed when the application terminates, you must close all KFP files within every program.

12.4.1 Keyed File Organization

A BASIC keyed file is organized using a data structure referred to as a *B-tree*. A B-tree consists of pages that are nodes in the tree. Within this B-tree structure are two different types of pages:

- Index pages
- Data pages

An index page contains only pointers to lower level pages (either index or data, but not both). A special index page, referred to as *page zero*, is always at the top level (root node) of the tree; all searches begin here. A data page always occupies a *leaf node* (the lowest level of the tree). The data page contains the actual logical data records. This type of data structure ensures efficient data access since the most recently used pages are retained in system memory. BASIC allows you to define the page size and the number of pages to be stored in memory. Depending on both the number of pages available to the KFP and which pages are being accessed, it is possible to read, write, or delete a record without a single disk access.

12.4.2 Keyed File Format

A keyed file is a relative record file whose special internal format allows keyed access. Each record in a keyed file can have 100 data fields of integer, real and/or string data type, including up to 10 variable-length key fields. You specify the data fields with a *record-format template* and the key fields with a *key field template* when the file is created.

The page size for a keyed file is equal to its physical record length (specified at creation) and is the minimum block of data transferred between disk and memory during a read or write operation. For optimum use of disk space, the page size should be a multiple of 256 bytes, which must be greater than or equal to a multiple of the minimum page size. To compute page size, use the following formula:

$$\text{PAGE SIZE} = (\text{MAXIMUM KEYED RECORD SIZE} * 2) + 6$$

The average length of a logical record, the maximum length of the composite key, and the total number of data records determine the number of ADUs required to hold the keyed file.

12.4.3 Keyed File Data Base Buffer Creation

Use the following procedure to create a keyed file buffer file:

1. Using the BLDBUF subroutine or the example utility (Figure 12-2), create a file of the desired buffer size. This file must be loaded as the first LIBRARY statement and must be preloaded by preceding the library name by an asterisk (*) in all programs using the keyed file package. This file is used as the buffer in all programs requiring a buffer of that size.
2. Execute the KFINIT subroutine to establish the parameters for the keyed file data base buffer. This subroutine must be run in every program that uses the KFP.

12.4.4 Keyed File Creation

You can create keyed files by using the KFCREA subroutine.

12.4.5 KFP Memory Management

You must allocate a certain amount of memory to the KFP for keyed file management functions. The amount of memory allocated depends on the number of subroutines simultaneously preloaded and the size of the keyed file data base buffer.

CAUTION

If, for any reason, the system becomes unusable while a KFP file is open, it should be assumed that file integrity is lost. It is the responsibility of the application programmer to restore lost files (using methods such as frequent back-ups and transaction logs).

12.4.5.1 Subroutine Memory Requirements. The KFP requires sufficient freespace to load the subroutines, as well as an additional contiguous block of memory at least as large as the page size of the keyed file. Memory requirements for each subroutine rounded to the next highest 50-byte increment are as follows:

Subroutine	Memory Requirement (in Bytes)
BLDBUF	1350
KFINIT	600
KFCREA	1650
KFOPEN	1150
KFCLOS	600
KFPUT	900
KFGET	700
KFREAD	2400
KFWRIT	3100
KFDELR	2150

You should never have an occasion to use the BLDBUF routine in an application, although you must run it before running an application. The other nine of the KFP subroutines require approximately 13,250 bytes of memory if they are preloaded. However, in most applications, KFINIT, KFCREA, KFOPEN, and KFCLOS need not be preloaded since they are rarely called. Only the frequently used subroutines (KFPUT, KFWRITE, KFGET, and KFREAD) need to be preloaded if they are needed for a particular application.

When writing a keyed file application program, the sum of the following items must not exceed the available user memory:

- Length of BASIC program
- Combined lengths of preloaded subroutines
- Length of largest nonpreloaded subroutine
- Length of KFP buffer
- Length of the additional contiguous page-sized memory block

If the KFP cannot obtain sufficient memory for any of the previous requirements, the application program terminates and the appropriate error message is returned.

12.4.5.2 Keyed File Data Base Buffer. The KFP contains a keyed file data base buffer created by the BLDBUF subroutine or by a utility similar to the one in Figure 12-2. The size of this buffer determines how many pages of a keyed file can be contained in memory at one time. Use the following equation to calculate the buffer size:

$$\text{BUFFER SIZE} = 26 + ((140 + K) * N) + (M * (P + 12))$$

where:

- K = $((R + F) + 1) \text{ AND } -2$
- R = record buffer size (maximum record size in bytes)
- F = maximum number of data fields
- N = number of files
- M = total number of pages in buffer
- P = page size (bytes)

The more pages of data in memory, the faster the keyed file can be searched for a specific logical record. However, including pages in memory reduces the amount of user freespace. Thus, you should carefully choose a buffer size that is large enough to permit efficient searching yet small enough to leave sufficient user freespace.

The KFINIT subroutine initializes the keyed file data base buffer according to the parameters you supply. These parameters determine the buffer's general operating characteristics. These parameters include:

- Length of data base buffer (stored within the buffer)
- Maximum number of keyed files open at one time
- Maximum page size
- Maximum number of pages simultaneously in memory (computed by KFINIT)
- Maximum data record size (record buffer)
- Maximum number of data fields (BASIC variables) in any record

12.4.5.3 Record Buffer. The data base buffer contains a record buffer for each KFP file. The record buffer serves as an intermediate area for the transfer of data to and from keyed files. Subroutines KFWRITE and KFREAD transfer data in the form of BASIC variables between this buffer and keyed files. KFWRITE transfers data from the buffer to the record, and KFREAD transfers data from the record to the buffer. The BASIC user program accesses the buffer by using subroutines KFGET and KFPUT. KFPUT transfers data to the buffer, and KFGET returns data from the buffer to the user program.

12.4.6 KFP Subroutines

All KFP file operations must be performed by a BASIC program. BASIC provides the following series of subroutines to access a keyed file:

Name	Description
BLDBUF	Build Keyed File Data Base Buffer
KFINIT	Initialize Keyed File Data Base Buffer
KFCREA	Create Keyed File
KFOPEN	Open Keyed File
KFPUT	Put Data into Record Buffer
KFWRITE	Write to Keyed File
KFREAD	Read from Keyed File
KFGET	Get Data from Record Buffer
KFDELR	Delete Keyed File Record
KFCLOS	Close a Keyed File

The following paragraphs describe each of the individual keyed file subroutines.

12.4.6.1 BLDBUF — Build Keyed File Data Base Buffer File. This subroutine builds the file to be loaded and used as the data base buffer by other keyed file package subroutines. The file created by this subroutine must be included as the first preloaded LIBRARY subroutine by all BASIC programs using the keyed file package. To execute this subroutine, use the following CALL statement:

```
CALL “.BLDBUF”(PATH$,BUFSIZ,REPLAC,STAT)
```

Argument details are summarized below:

Argument	Type	I/O	Use
PATH\$	String	Input	Pathname of buffer file
BUFSIZ	Integer	Input	Computed data base buffer size
REPLAC	Integer	Input	Output file deletion code*
STAT	Integer	Output	Subroutine error status

* Argument notes and special considerations:

REPLAC This integer code specifies whether the buffer file will be replaced if it already exists. The following list shows the possible values for this argument:

0 — An existing file of the same name will not be replaced.

1 — An existing file of the same name will be replaced.

12.4.6.2 KFINIT — Initialize Keyed File Data Base Buffer. This subroutine establishes the general parameters for using keyed files on BASIC. Until this subroutine is run, keyed files cannot be accessed or manipulated. This subroutine can also be used to modify KFP parameters previously established. It cannot be run if a keyed file is currently open.

The KFINIT subroutine sets up the KFP parameters within the keyed file data base buffer. Note that you must use the BLDBUF subroutine to define the keyed file data base buffer before you can execute any other KFP subroutine. KFINIT allows you to specify the following parameters:

- Maximum page size
- Maximum number of open keyed files
- Largest data record allowed
- Maximum number of data fields per record reserved per file

If the buffer is large enough to contain at least two pages, then the buffer is initialized based on the supplied input parameters. If there is insufficient buffer space to contain at least two pages, then the subroutine returns an error, and the buffer is not initialized. In either case, the number of pages contained in the buffer is returned to the calling program in the third argument (NUMPG).

To execute this subroutine, use the following CALL statement:

```
CALL ".KFINIT"(DUMY,STAT,NUMPG,MAXPGZ,RECSIZ,NUMFLD,NUMFIL)
```

Argument details are summarized below:

Argument	Type	I/O	Use
DUMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
NUMPG	Integer	Output	Number of pages in memory*
MAXPGZ	Integer	Input	Maximum size of KFP memory page*
RECSIZ	Integer	Input	Size of largest data record*
NUMFLD	Integer	Input	Maximum number of fields in data record*
NUMFIL	Integer	Input	Maximum number of open keyed files*

* Argument notes and special considerations:

- NUMPG This output parameter specifies the exact number of pages of a specified size in the keyed file data base buffer. A keyed file is accessed most rapidly when this value is large (for example, four or more pages in memory per file).
- MAXPGZ The maximum page size must be at least as large as the largest physical record (page) size used by any keyed file. This parameter must be expressed in bytes. The minimum page size is 18 bytes. This value cannot be less than any PGSIZ argument used by the KFCREA subroutine.
- RECSIZ This parameter specifies the largest data record that can be read or written to a keyed file. This value cannot be greater than half the largest physical record size minus six bytes. This value specifies the size of the record buffer contained in the keyed file data base buffer. The minimum data record size is two bytes. Note that the value must include a length byte for each string included in the buffer.
- NUMFLD This parameter specifies the maximum number of data fields that can be contained in any record. A data field is specified by a BASIC variable and can be an integer, string, or real value. This value must be at least 1, cannot be greater than half the data record size, and cannot be greater than 100.
- NUMFIL This argument specifies the maximum number of open keyed files allowed at one time. This value must be at least 1 and not more than 15.

12.4.6.3 KFCREA — Create Keyed File. This subroutine creates a keyed file. To execute this subroutine, use the following CALL statement:

```
CALL ".KFCREA"(DUMY,STAT,PATH$,PGSIZ,MAXKEY,ALLOC,RFT$,KFT$)
```

Argument details are summarized below:

Argument	Type	I/O	Use
DUMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
PATH\$	String	Input	Pathname of keyed file
PGSIZ	Integer	Input	Physical record size (page size)*
MAXKEY	Integer	Input	Maximum size of composite key*
ALLOC	Integer	Input	Primary allocation in physical records
RFT\$	String	Input	Record-format template*
KFT\$	String	Input	Key-field template*

* Argument notes and special considerations:

- PGSIZ** This argument determines the file's page size (physical record size) and must not exceed the maximum page size specified in the KFINIT subroutine. Determination of the page size is based largely on data record size and number of records per page. A large page size reduces disk I/O but requires more memory. The KFP performs physical I/O operations on pages rather than on individual records.
- MAXKEY** This parameter specifies the maximum size (in bytes) of the composite key. The composite key is the concatenation of all the key fields. Refer to the SIZ description in the SORT subroutine for information on calculating this value. This value determines the number of bytes reserved in the index record for each composite key. This value should be as small as possible to ensure the most efficient loading of index pages. The value cannot exceed $((\text{PAGE SIZE} - 6)/2) - 2$.
- ALLOC** Primary allocation in physical records. You can pass a value of zero for the system default.
- RFT\$** The record-format template describes the arrangement of data types (BASIC variables) within a keyed record. The record-format template is a string with the following format:

TYPE1,TYPE2,...TYPE_n

where the type equals one of the following:

I = Integer data

R = Real data

\$ = String data

Consecutive fields of the same type can be designated by preceding an identifier with n*, where n is an integer that indicates the number of consecutive fields of the same data type. For example, the string \$,2*I,R,2*\$ specifies a record containing a string, followed by two integers, followed by a real value, followed by two strings. The number of fields defined must not exceed the maximum number of fields defined at data base buffer initialization.

KFT\$ This parameter defines the key-field template that specifies which data fields are to be used for the keys. It is a multiple numeric index to the data fields. Up to ten data fields can be used as key fields. The key-field template is a string variable with the following format:

KEYFLD1:< UP/DN> ,KEYFLD2:< UP/DN> ,...,KEYFLDn:< UP/DN>

where:

KEYFLD1...KEYFLDn are integer values indexing the data fields

A colon separates the KEYFLDn integer value from the UP or DN string

UP or DN are strings that indicate whether the key field is to be accessed in ascending or descending order

For example, if the third and fifth data fields are key fields and are to be accessed in ascending order, the key-field template should be defined by the following:

KFT\$ = "3:UP,5:UP"

The KFP subroutines search for records using the key fields. They compare the key fields defined by the key-field template (KFT\$) with the key fields in the record. If the first search key field matches the first record key field, the second fields are compared, and so on, until two unequal fields are found or until all key fields have been compared.

12.4.6.4 KFOPEN — Open Keyed File. This subroutine opens a keyed file for access by a user program. It operates in a manner similar to the BASIC OPEN statement except that the system rather than the user assigns the unit number. The unit number must be referenced when performing any I/O on the file.

To execute this subroutine, use the following CALL statement:

CALL ".KFOPEN"(DUMMY,STAT,PATH\$,UNITNO,ACCESS)

Argument details are summarized below:

Argument	Type	I/O	Use
DUMMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
PATH\$	String	Input	Pathname of file to open
UNITNO	Integer	Output	Unit number assigned to keyed file*
ACCESS	Integer	Input	Specifies access privileges*

* Argument notes and special considerations:

UNITNO The unit number can range from 1 to 15.

ACCESS Specifies access privileges; the default is read/write without file sharing. If you want the file to be read only, and you want multiple users to be able to read the file at one time, you must specify the access privileges. If `ACCESS = 0`, or if you do not specify the access parameter, then the access privilege is read/write. When `ACCESS` is not equal to zero, access privilege is read only.

12.4.6.5 KFPUT — Put Data into Record Buffer. This subroutine inserts data in the form of BASIC variables into the record buffer. The record buffer can then be written to a record of a specified keyed file by using the `KFWRITE` subroutine. You specify whether the record buffer is initialized (cleared) before the insertion of variables.

To execute this subroutine, use the following `CALL` statement:

```
CALL ".KFPUT"(DUMMY,STAT,UNITNO,INIT,FLDNUM,DATA1,...DATAn)
```

where:

$1 < n < = 10$

Argument details are summarized below:

Argument	Type	I/O	Use
DUMMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
UNITNO	Integer	Input	Keyed file unit number
INIT	Integer	Input	Record buffer initialization code: 0 = Do not initialize 1 = Initialize
FLDNUM	Integer	Input	Starting field number*
DATA1	Any	Input	Optional first data argument*
.	.	Input	.
.	.	Input	.
.	.	Input	.
DATAn	Any	Input	Optional nth data argument* (where $1 < n <= 10$)*

* Argument notes and special considerations:

- INIT** This argument specifies whether the record buffer is initialized before access by the KFPUT subroutine. The record buffer should be initialized only once after opening the keyed file. Note that the record buffer need not be initialized if a read operation is performed before executing the KFPUT subroutine.
- FLDNUM** This argument specifies the field number at which the KFPUT subroutine places DATA1 (data argument one). Any subsequent data arguments (DATA2...DATAn) are sequentially inserted in the following fields in the record buffer. Insertion can begin with any field in the record, but DATA1...DATAn must correspond to the record-format template from FLDNUM forward. Note that if FLDNUM plus n is larger than the total number of fields in the record, a record-buffer-overflow error occurs. If overflow occurs, none of the data arguments are placed into the record buffer.
- DATAn** The data arguments are used to pass data to the record buffer from the BASIC program. If no data arguments are specified and the INIT argument is set equal to 1, the subroutine only initializes the record buffer.

12.4.6.6 KFWRITE — Write Keyed File. This subroutine writes the contents of the record buffer to a keyed file. The subroutine uses the keys specified in the buffer to search for the appropriate record within the file. To execute this subroutine, use the following CALL statement:

```
CALL ".KFWRITE"(DUMMY,STAT,UNITNO,IMMED,REP)
```

Argument details are summarized below:

Argument	Type	I/O	Use
DUMMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
UNITNO	Integer	Input	Unit number assigned to keyed file
IMMED	Integer	Input	Immediate write option code:* 0 = No immediate write 1 = Immediate write
REP	Integer	Input	Record replace code:* 0 = Do not replace record 1 = Replace record

* Argument notes and special considerations:

IMMED	This argument specifies whether the write operation causes a page to be written to disk immediately. If you select the immediate option, the subroutine updates records on disk as soon as the modification is made. Otherwise, the subroutine does not update files on disk until a page is forced back to disk or the file is closed. Using the immediate option can result in slower subroutine execution times due to frequent disk I/O.
REP	If this argument equals 0, the subroutine writes the record into the file only if it does not already exist. If this argument equals 1, the subroutine writes the record to the file whether or not a record with the specified key already exists.

12.4.6.7 KFREAD — Read Keyed File. This subroutine reads a specified record from a keyed file into the record buffer. You specify whether to search the file for a record specified by a unique key value or to search the file sequentially for the next higher or next lower record. You can access the data contained in the record buffer by calling the KFGET subroutine. To execute this subroutine, use the following CALL statement:

```
CALL ".KFREAD"(DUMMY,STAT,UNITNO,OP,KEY1,< KEY2,KEY3,...KEYn>)
```

Argument details are summarized below:

Argument	Type	I/O	Use
DUMMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
UNITNO	Integer	Input	Unit number assigned to keyed file
OP	Integer	Input	Operation code for subroutine:* 0 = Find record equal to key 1 = Find record less than or equal to key 2 = Find record greater than or equal to key 3 = Find next record less than key 4 = Find next record greater than key
KEY1	Any	Input	Required first key value
KEY2	Any	Input	Optional second key value
.	.	.	.
.	.	.	.
KEYn	Any	Input	Optional nth key value (n< = 10)

* Argument notes and special considerations:

OP This argument specifies whether the subroutine searches the file for a record with a specific key value or searches the file for a record with the next lower or next higher key value. If this argument equals 0, the subroutine reads the record with the specified key value. If this argument equals 1, the subroutine reads the record with the specified key or the next lower key if the record does not exist. If this argument equals 2, the subroutine reads the record with the specified key or the next higher key if the record does not exist. If the argument equals 3, the subroutine reads the record with the next lower key value. If the argument equals 4, the subroutine reads the record with the next higher key value.

12.4.6.8 KFGET — Get Data from Record Buffer. This subroutine returns data in the form of BASIC variables from the record buffer to the calling program. This subroutine is used in conjunction with the KFREAD subroutine to read the contents of keyed files. You must specify the starting variable field number to be returned. A maximum of 10 data arguments can be returned per call, and the data may be returned in nonsequential order. To execute this subroutine, use the following CALL statement:

```
CALL ".KFGET"(DUMY,STAT,UNITNO,FLDNUM,< DATA1,...DATAn> )
```

where:

$$1 < n < = 10$$

Argument details are summarized below:

Argument	Type	I/O	Use
DUMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
UNITNO	Integer	Input	Unit number of keyed file
FLDNUM	Integer	Input	Starting field number
DATA1	Any	Output	Optional first data argument
.	.	.	.
.	.	.	.
.	.	.	.
DATAn	Any	Input	Optional nth data argument*

* Argument notes and special considerations:

DATAn The data arguments are used to return data from the record buffer. The argument must be of the same data type as the BASIC variable data contained in the record buffer as defined in the record-format template. If the subroutine is used to return string data, the calling arguments must be large enough to contain the data. The argument should be initialized either to the length equal to or greater than that of the data. If the argument's length is less than that of the data, the former is filled with asterisks. If the argument's length is greater than that of the data, the former contains the data and additional blanks to pad the remainder of the string. If you have no indication of the length of the data string, you should use an argument initialized to 255 characters.

The data arguments (DATA1 through DATAn) cannot be initialized to a zero-length string, even though a zero-length string may exist in the record buffer. Data arguments are filled with blanks if a corresponding element in the record buffer has a zero length.

12.4.6.9 KFDELRL — Delete Keyed File Record. This subroutine deletes a record within a keyed file. You must specify the keyed file unit number and the key value(s) of the record to be deleted. You can specify whether the record is deleted immediately or at page replacement or close. To execute this subroutine, use the following CALL statement:

CALL “.KFDELRL”(DUMY,STAT,UNITNO,IMMED,KEY1,< KEY2,KEY3,....,KEYn>)

Argument details are summarized below:

Argument	Type	I/O	Use
DUMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
UNITNO	Integer	Input	Unit number of keyed file
IMMED	Integer	Input	Delete immediate code:* 0 = No immediate delete 1 = Immediate delete
KEY1	Any	Input	Required first key value
KEY2	Any	Input	Optional second key value
.	.	.	.
.	.	.	.
.	.	.	.
KEYn	Any	Input	Optional nth key value (n< = 10)

* Argument notes and special considerations:

IMMED This argument specifies whether the delete operation causes a page to be written to disk. If you select the immediate delete option, the subroutine deletes the record from the disk immediately. If you do not select the immediate delete option, records are not deleted until a page is forced back to disk or the file is closed. Using the immediate option can result in slower subroutine execution times due to frequent disk I/O.

12.4.6.10 KFCLOS — Close Keyed File. This subroutine closes a specified keyed file. It operates in a manner similar to that of the BASIC CLOSE statement. After a file is closed, it is unavailable for access (either read or write) until the KFOPEN subroutine opens it. If you did not select the immediate write option in the KFWRITE subroutine, the KFCLOS subroutine writes all pages containing modified records to the disk before closing. To execute this subroutine, use the following CALL statement:

```
CALL ".KFCLOS"(DUMMY,STAT,UNITNO)
```

Argument details are summarized below:

Argument	Type	I/O	Use
DUMMY	Integer	Input	Reserved for future use
STAT	Integer	Output	Subroutine error status
UNITNO	Integer	Input	Unit number of file to be closed

12.4.7 Keyed File Example

Figure 12-2 illustrates the use of the KFP subroutines in a BASIC program. This example allows you to build a keyed file containing names and telephone numbers. You access the phone numbers by name.

To build the keyed file application, the following assumptions were made about the data:

- The phone directory will contain a maximum of 100 entries (records).
- Each entry (record) will consist of a last name and phone number.
 - The last name will be the key field and can be up to 40 characters (bytes) in length. It is a string data item with a length of 41 bytes (40 character bytes plus one length byte).
 - The phone number will be a data item and can be up to 14 characters (bytes) in length. It is a string data item with a length of 15 bytes (14 character bytes plus one length byte).

You must set up the KFP data base buffer, either by using the BLDBUF subroutine or by using a utility such as the one in Figure 12-2. The following formula is used to calculate the size of the buffer. To keep the size of the buffer small, the number of pages contained within the buffer (M) has been set to the minimum value of two.

$$\text{BUFFER SIZE} = 26 + ((140 + K) * N) + (M * (P + 12))$$

where:

- K = $((R + F) + 1) \text{ AND } -2 = ((56 + 2) + 1) \text{ AND } -2 = 58$ bytes
- R = record buffer size = maximum record size = 56 bytes
- F = maximum number of data fields = 2
- N = number of files = 1
- M = number of pages in buffer = 2
- P = page size = 256 bytes

$$\text{BUFFER SIZE} = 26 + ((140 + 58) * 1) + (2 * (256 + 12)) = 760 \text{ bytes}$$

Enter the value 760 in response to the BUFFER SIZE prompt of the sample Build Buffer utility in Figure 12-2; note that in Figure 12-2, KFP is a synonym for the directory containing the KFP subroutines.

After you build the KFP buffer (Figure 12-2), the system can run the following program (Figure 12-3).

```

100 INTEGER ALL
110 LIBRARY "*KFP.BLDBUF"
120 DISPLAY ERASE ALL AT(1,1)"CSD BASIC KFP BUFFER UTILITY"
130 DISPLAY AT(2,8)"FILE PATHNAME: .KFPBUF"
140 DISPLAY AT(3,8)" BUFFER SIZE: "
150 DISPLAY AT(4,8)"REPLACE(Y/N)? : N"
160 DISPLAY AT(5,8)" OK(Y/N/Q)? : "
170 C=23
180 DISPLAY AT(24,1)
190 ACCEPT AT(2,C)SIZE(-48)"" :P$
200 ACCEPT AT(3,C)SIZE(-6)"" :SZ$
210 IF NOT NUMERIC(SZ$) THEN 200
220 IF VAL(SZ$)>32766 THEN 200 ELSE SZ=VAL(SZ$)
230 SZ=((SZ+1) AND -2)
240 DISPLAY AT(3,C)STR$(SZ)
250 ACCEPT AT(4,C)SIZE(-3)"" :RP$
260 RP=POS("NY",SEG$(RP$,1,1),1)
270 IF RP=0 THEN 250 ELSE RP=RP-1
280 ACCEPT AT(5,C)SIZE(-3)"" :OK$
290 OK=POS("NYQ",SEG$(OK$,1,1),1)+1
300 ON OK GOTO 280,180,320,310
310 STOP
320 ER=0
330 DISPLAY AT(24,1)"== UTILITY EXECUTING =="
340 CALL "KFP.BLDBUF"(P$,SZ,RP,ER)
350 DISPLAY AT(24,1)
360 IF ER=0 THEN DISPLAY AT(24,1)"** UTILITY COMPLETE **";: GOTO 400
370 IF ER>=1000 THEN DXER$="(DX "&HEXASC$(ER-1000)&)" ELSE DXER$=""
380 EM$="** ERROR RETURNED: #"&STR$(ER)&" "&DXER$&" **"
390 DISPLAY AT(24,1)EM$;
400 DISPLAY BELL; : : A$=INKEY$(0)
410 IF ASC(A$)=141 OR ASC(A$)=160 THEN 180
420 IF ASC(A$)=155 THEN 310 ELSE 400
430 !
440 DEF HEXASC$(I)
450 DIM CN$(7)::CN$(0)="8"::CN$(1)="9"
455 FOR W=2 TO 7::CN$(W)=CHR$(W+63)::NEXT W
460 IF I<0 THEN FLG=-1::Z=I AND 32767 ELSE Z=I::FLG=0
470 H$=""
480 FOR J=1 TO 4
490 X=INT(Z/16) :: Y=Z-(X*16) :: Z=X
500 IF Y<10 THEN Y=Y+48 ELSE Y=Y+55
510 H$=CHR$(Y)&H$
520 NEXT J
530 IF FLG THEN HEXASC$=">"&CN$(VAL(SEG$(H$,1,1))&SEG$(H$,2,3)::GOTO 550
540 HEXASC$=">"&H$
550 FNEND

```

Figure 12-2. Sample Build Buffer Utility

```

100 INTEGER ALL :: LIBRARY "*KFP.KFPBUF"
110 LIBRARY "KFP.KFOPEN","KFP.KFINIT","KFP.KFCREA","*KFP.KFPUT","*KFP.KFWRIT"
120 LIBRARY "*KFP.KFREAD","*KFP.KFGET","KFP.KFDELRL","KFP.KFCLOS"
130 !***** INITIALIZE KFP *****
140 !
150 DUMY=1    :: STAT=0      :: NUMPG=0      :: MAXPGZ=256
160 RECSIZ=56:: NUMFLD=2    :: NUMFIL=1
170 CALL "KFP.KFINIT"(DUMY,STAT,NUMPG,MAXPGZ,RECSIZ,NUMFLD,NUMFIL)
180 IF STAT>0 THEN DISPLAY "KFINIT status = ";STAT :: STOP
190 !***** CREATE/OPEN FILE *****
200 !
210 DUMY=1    :: STAT=1      :: PATH$="KFP.FILEP"  :: PGSIZ=256  :: ACCESS = 0
220 MAXKEY=41 :: ALLOC=35    :: RFT$="$,$"    :: KFT$="1:UP"
230 ACCEPT"DO YOU WANT TO CREATE THE DIRECTORY?(Y/N)":CR$
240 IF CR$="Y" THEN 250 ELSE 350
250 PRINT ERASE ALL
260 PRINT "The number of pages in the data base buffer = "; NUMPG
270 PRINT
280 PRINT " If the number is not optimal, modify the "
290 PRINT " data base buffer size using the BLDBUF subroutine"
300 PRINT
310 ACCEPT "*** Press RETURN to Continue ***":CR$
320 DEL PATH$
330 CALL"KFP.KFCREA"(DUMY,STAT,PATH$,PGSIZ,MAXKEY,ALLOC,RFT$,KFT$)
340 IF STAT>0 THEN DISPLAY "KFCREA status = ";STAT :: STOP
350 CALL "KFP.KFOPEN"(DUMY,STAT,PATH$,UNITNO,ACCESS)
360 IF STAT>0 THEN DISPLAY "KFOPEN status = ";STAT :: STOP
370 !***** MENU SECTION *****
380 !
390 DISPLAY ERASE ALL
400 DISPLAY AT(5,15)"          PHONE NUMBER DIRECTORY - Menu"
410 DISPLAY AT(7,20)"          A - ADD ENTRY  "
420 DISPLAY AT(8,20)"          D - DELETE ENTRY "
430 DISPLAY AT(9,20)"          F - FIND NUMBER  "
440 DISPLAY AT(10,20)"         E - EXIT DIRECTORY"
450 DISPLAY
460 ACCEPT AT(15,20)" ENTER CODE FROM ABOVE MENU ":C$
470 IF C$="A" THEN 580
480 IF C$="D" THEN 700
490 IF C$="F" THEN 780
500 IF C$="E" THEN 530 ELSE GOTO 390
510 !***** CLOSE FILE *****
520 !
530 CALL "KFP.KFCLOS"(DUMY,STAT,UNITNO)
540 IF STAT>0 THEN DISPLAY "KFCLOS status = ";STAT :: STOP
550 STOP
560 !***** WRITE RECORDS SECTION *****
570 !
580 DISPLAY ERASE ALL
590 ACCEPT "ENTER THE LAST NAME   ":NAME$
600 ACCEPT "ENTER THE PHONE NUMBER ":NUMBER$
610 INIT=1 :: FLDNUM=1
620 CALL "KFP.KFPUT"(DUMY,STAT,UNITNO,INIT,FLDNUM,NAME$,NUMBER$)
630 IF STAT>0 THEN DISPLAY "KFPUT status = ";STAT :: STOP
640 IMMED=0 :: REP=1
650 CALL "KFP.KFWRIT"(DUMY,STAT,UNITNO,IMMED,REP)
660 IF STAT>0 THEN DISPLAY "KFWRIT status = ";STAT :: STOP
670 GOTO 390
680 !***** DELETE RECORDS SECTION *****
690 !
700 IMMED=0

```

Figure 12-3. Keyed File Example (Sheet 1 of 2)

```
710 ACCEPT "ENTER THE NAME TO BE DELETED ":NAME$
720 CALL "KFP.KFDEL" (DUMY,STAT,UNITNO,IMMED,NAME$)
730 IF STAT=377 THEN DISPLAY "RECORD DOES NOT EXIST " :: GOTO 880
740 IF STAT>0 THEN DISPLAY "KFDEL status = ";STAT :: STOP
750 GOTO 390
760 !***** READ RECORDS SECTION *****
770 !
780 NAME$=" " :: ACCEPT "ENTER THE LAST NAME " :NAME$
790 OP=0 :: NUMBER$= " "
800 CALL "KFP.KFREAD" (DUMY,STAT,UNITNO,OP,NAME$)
810 IF STAT=377 THEN DISPLAY "RECORD DOES NOT EXIST" :: GOTO 880
820 IF STAT>0 THEN DISPLAY "KFREAD status =";STAT :: STOP
830 NUMBER$=" " :: FLDNUM=1
840 CALL "KFP.KFGET" (DUMY,STAT,UNITNO,FLDNUM,NAME$,NUMBER$)
850 IF STAT>0 THEN DISPLAY "KFGET status =";STAT :: STOP
860 DISPLAY ERASE ALL
870 DISPLAY AT(10,20) NAME$;" ";NUMBER$
880 ACCEPT "press RETURN to continue":A$
890 GOTO 390
900 STOP
```

Figure 12-3. Keyed File Example (Sheet 2 of 2)

Appendix A

Keycap Cross-Reference

Generic keycap names that apply to all terminals are used for keys on keyboards throughout this manual. This appendix contains specific keyboard information to help you identify individual keys on any supported terminal. For instance, every terminal has an Attention key, but not all Attention keys look alike or have the same position on the keyboard. You can use the terminal information in this appendix to find the Attention key on any terminal.

The terminals supported are the 931 VDT, 911 VDT, 915 VDT, 940 EVT, the Business System terminal, and hard-copy terminals (including teleprinter devices). The 820 KSR has been used as a typical hard-copy terminal. The 915 VDT keyboard information is the same as that for the 911 VDT except where noted in the tables.

Appendix A contains three tables and keyboard drawings of the supported terminals.

Table A-1 lists the generic keycap names alphabetically and provides illustrations of the corresponding keycaps on each of the currently supported keyboards. When you need to press two keys to obtain a function, both keys are shown in the table. For example, on the 940 EVT the Attention key function is activated by pressing and holding down the Shift key while pressing the key labeled PREV FORM NEXT. Table A-1 shows the generic keycap name as Attention, and a corresponding illustration shows a key labeled SHIFT above a key named PREV FORM NEXT.

Function keys, such as F1, F2, and so on, are considered to be already generic and do not need further definition. However, a function key becomes generic when it does not appear on a certain keyboard but has an alternate key sequence. For that reason, the function keys are included in the table.













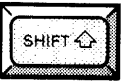
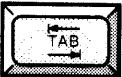





























Multiple key sequences and simultaneous keystrokes can also be described in generic keycap names that are applicable to all terminals. For example, you use a multiple key sequence and simultaneous keystrokes with the log-on function. You log on by *pressing the Attention key, then holding down the Shift key while you press the exclamation (!) key*. The same information in a table appears as *Attention/(Shift)!*.

Table A-2 shows some frequently used multiple key sequences.

Table A-3 lists the generic names for 911 keycap designations used in previous manuals. You can use this table to translate existing documentation into generic keycap documentation.

Figures A-1 through A-5 show diagrams of the 911 VDT, 915 VDT, 940 EVT, 931 VDT, and Business System terminal, respectively. Figure A-6 shows a diagram of the 820 KSR.

Table A-1. Generic Keycap Names

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820 ¹ KSR
Alternate Mode	None				None
Attention ²		 			 
Back Tab	None	 	 	None	 
Command ²					 
Control					
Delete Character					None
Enter					 
Erase Field					 

Notes:

¹The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

²On a 915 VDT the Command Key has the label F9 and the Attention Key has the label F10.

2284734 (2/14)




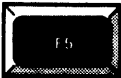


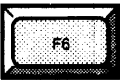








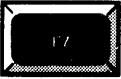


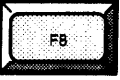










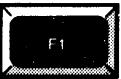










Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820 ¹ KSR
Erase Input					
Exit			 	 	
Forward Tab	 			 	
F1					
F2					
F3					
F4					

Notes:

¹The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.














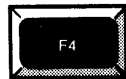




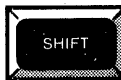














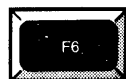









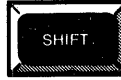





Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820' KSR
F5					 
F6					 
F7					 
F8					 
F9	 			 	 
F10	 			 	 

Notes:

*The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820' KSR
F11	 			 	 
F12	 			 	 
F13	 	 	 	 	 
F14	 	 	 	 	 
Home					 
Initialize Input		 			 

Notes:

*The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.














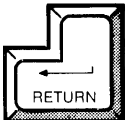




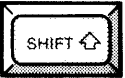



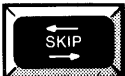







Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820' KSR
Insert Character					None
Next Character	 or 				None
Next Field	 		 	 	None
Next Line					 or
Previous Character	 or 				None
Previous Field		 			None

Notes:

*The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

Table A-1. Generic Keycap Names (Continued)

Generic Name	911 VDT	940 EVT	931 VDT	Business System Terminal	820 ¹ KSR
Previous Line					 
Print					None
Repeat		See Note 3	See Note 3	See Note 3	None
Return					
Shift					
Skip					None
Uppercase Lock					

Notes:

¹The 820 KSR terminal has been used as a typical hard-copy terminal with the TPD Device Service Routine (DSR). Keys on other TPD devices may be missing or have different functions.

²The keyboard is typamatic, and no repeat key is needed.

228 47 3 4 (7/14)

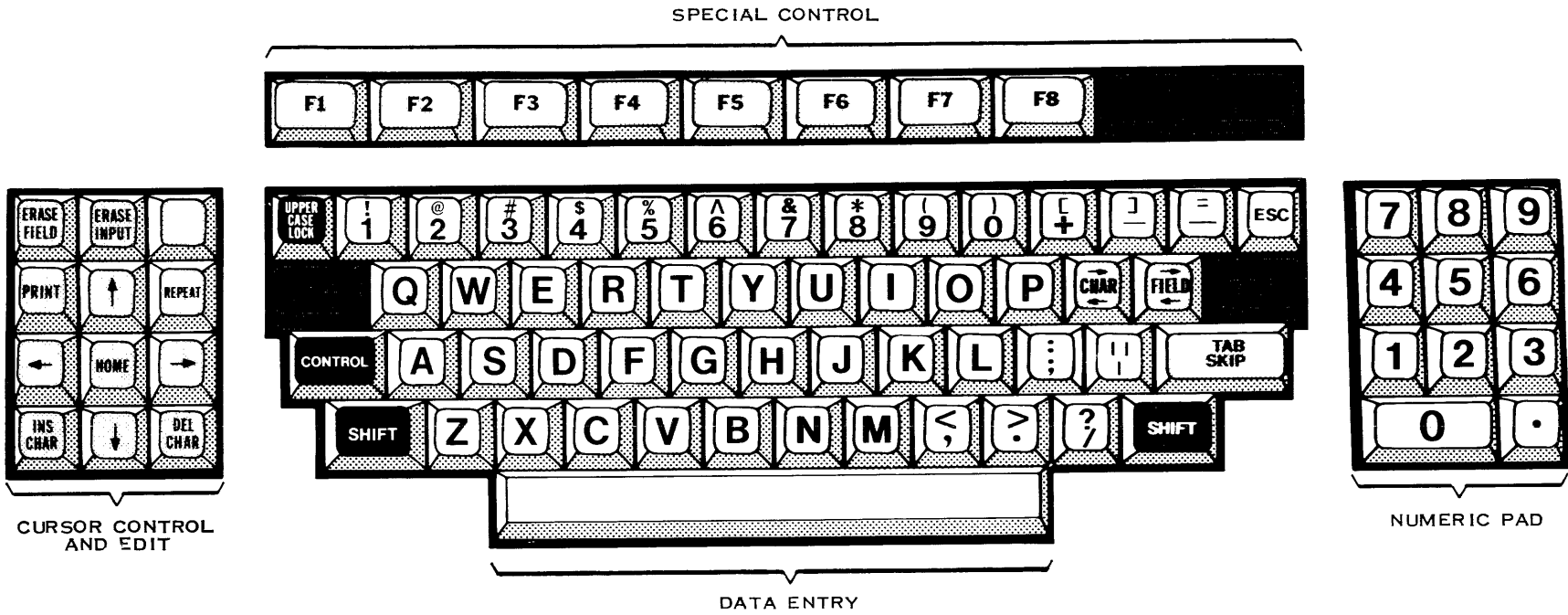
Table A-2. Frequently Used Key Sequences

Function	Key Sequence
Log-on	Attention/(Shift)!
Hard-break	Attention/(Control)x
Hold	Attention
Resume	Any key

Table A-3. 911 Keycap Name Equivalents

911 Phrase	Generic Name
Blank gray	Initialize Input
Blank orange	Attention
Down arrow	Next Line
Escape	Exit
Left arrow	Previous Character
Right arrow	Next Character
Up arrow	Previous Line

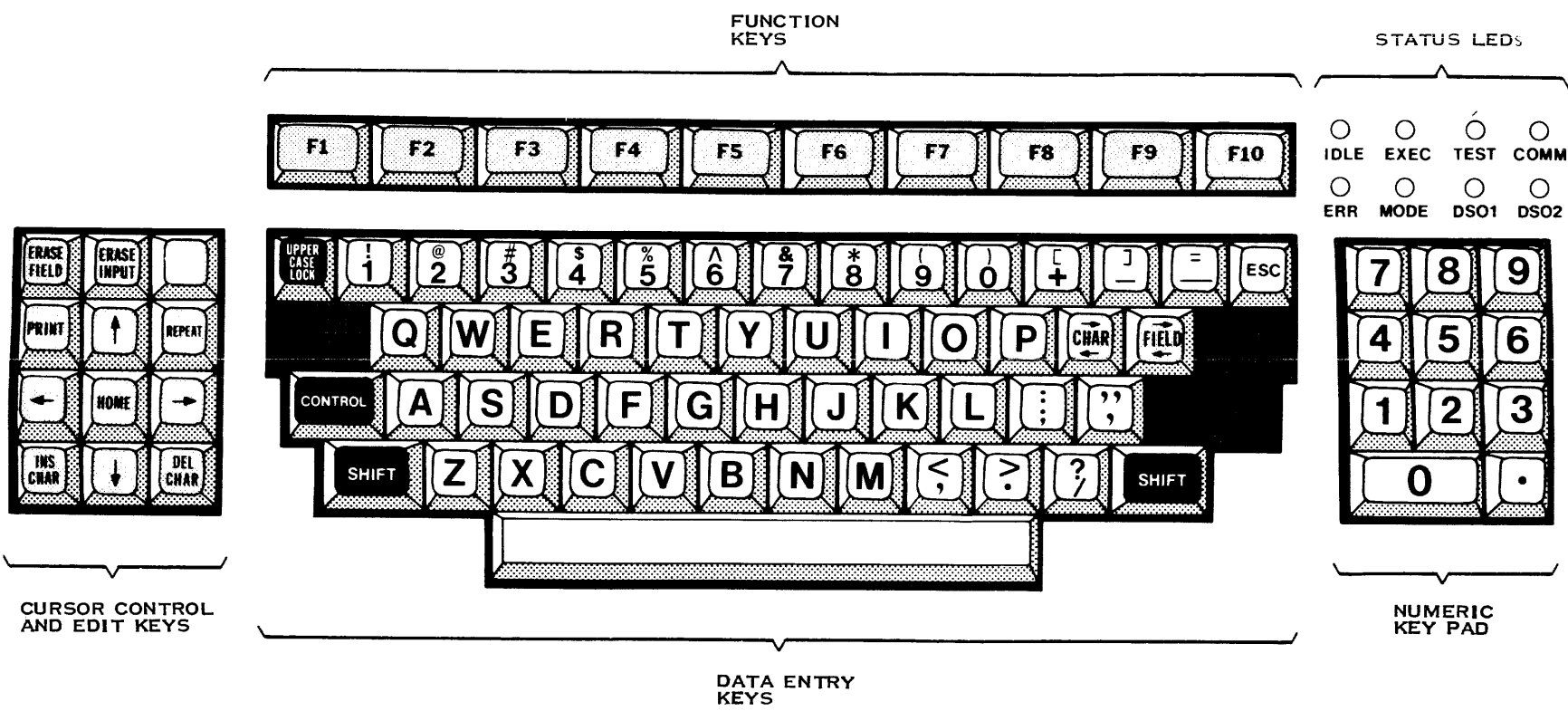
2.284734 (8/14)



2284734 (9/14)

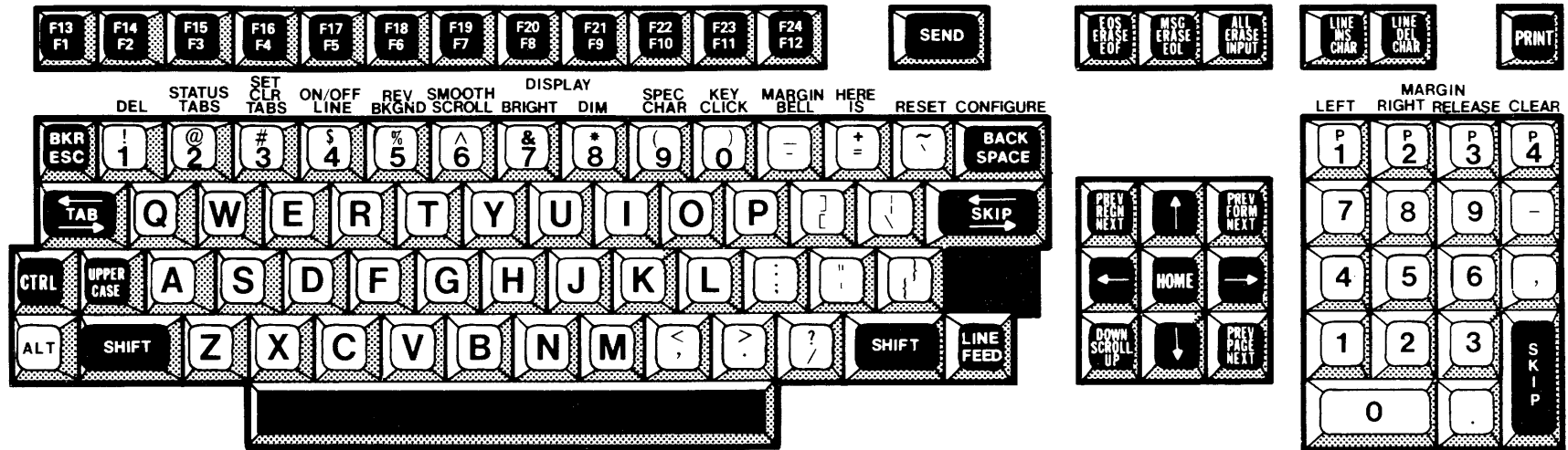
Figure A-1. 911 VDT Standard Keyboard Layout

Change 1



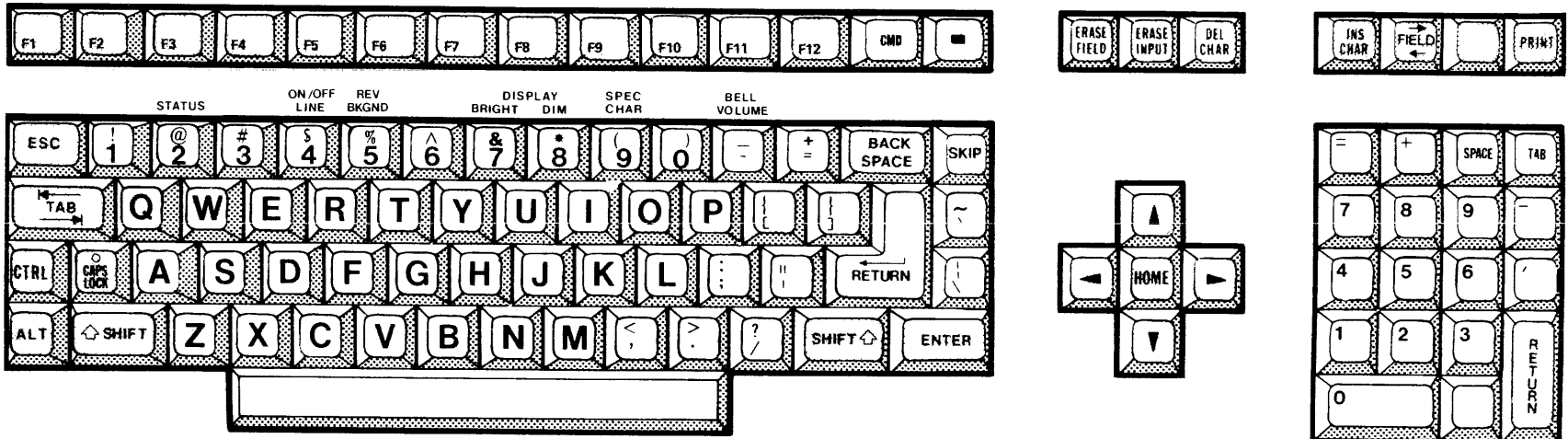
2284734 (10/14)

Figure A-2. 915 VDT Standard Keyboard Layout



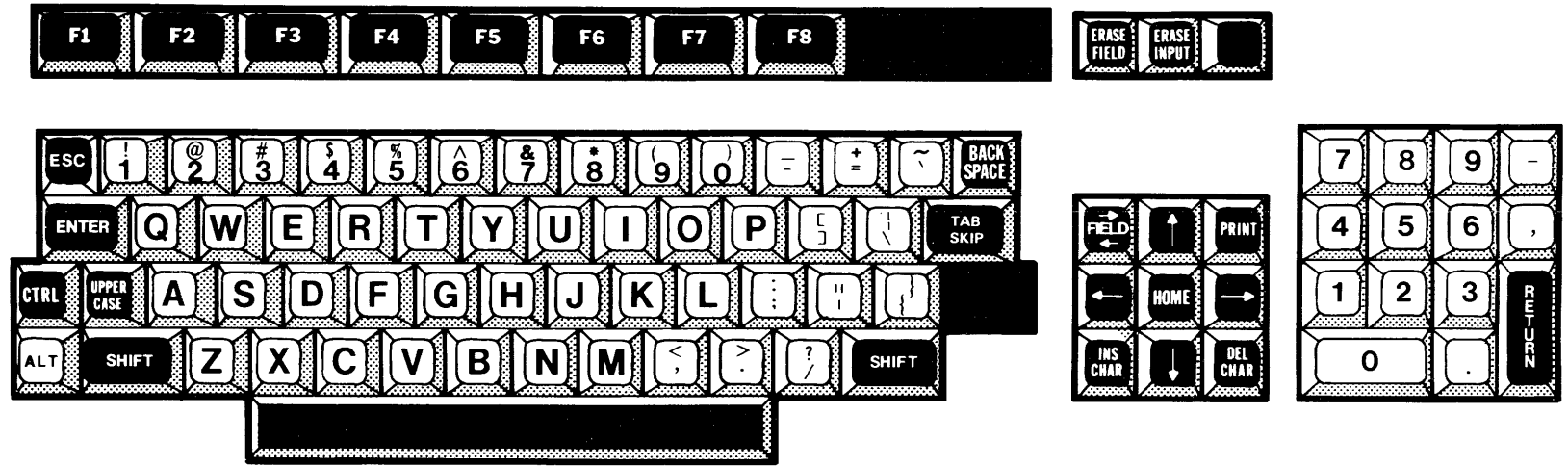
2284734 (11/14)

Figure A-3. 940 EVT Standard Keyboard Layout



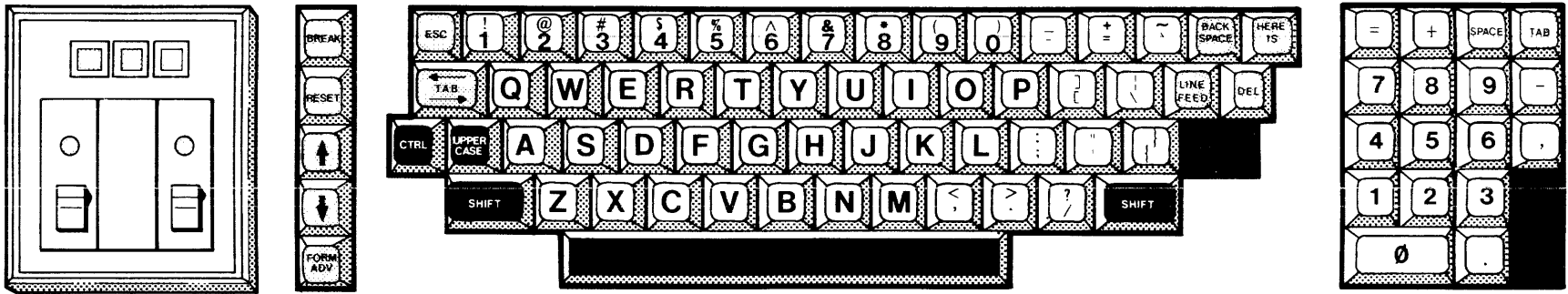
2284734 (12/14)

Figure A-4. 931 VDT Standard Keyboard Layout



2284734 (13/14)

Figure A-5. Business System Terminal Standard Keyboard Layout



2284734 (14/14)

Figure A-6. 820 KSR Standard Keyboard Layout

Appendix B

ASCII Character Set

Table B-1 lists the ASCII characters and graphics characters supported by TI BASIC. The ASCII value is either the value returned when the convert-ASCII-to-decimal function (ASC) is applied to the character or the argument of the CHR\$ function that returns the corresponding character.

In BASIC, the codes 7, 10, and 13 produce control actions instead of graphics. Adding 128 to these values results in graphics. BASIC subtracts 128 from the codes greater than 127 to determine which graphic to display. The statement PRINT CHR\$(I) following tables when I has the value indicated in the I-code column. You can determine the characters that can be displayed on a given system by entering the following program:

```
FOR I = 0 TO 255 :: PRINT I;CHR$(I) :: NEXT I
```

Table B-1 shows the output of this program. In the above program, the output is indexed by the value of I for easy reference.

NOTE

Graphics characters are not displayed on the screen for BASIC for the following codes:

Code	Description
07	The beeper is sounded and nothing is displayed.
10	Nothing is displayed.
13	A new line is started.

Table B-1. ASCII Character/Graphic Character Codes

I CODE	ASCII CHARACTER*	DISPLAYED CHARACTER	I CODE	DISPLAYED CHARACTER
00	NUL	-	32	SPACE
01	SOH	▪	33	!
02	STX	▪	34	"
03	ETX	▪	35	#(£ FOR U.K.)
04	EOT	▪	36	\$(¤ SWEDISH)
05	ENQ	▪	37	%
06	ACK	▪	38	&
07	BEL	▪	39	'
08	BS	▪	40	(
09	HT	▪	41)
10	LF	▪	42	◀
11	VT	▪	43	+
12	FF	▪	44	,
13	CR*	▪	45	-
14	SO	▪	46	.
15	SI	▪	47	/
16	DLE	▪	48	0
17	DC1	▪	49	1
18	DC2	▪	50	2
19	DC3	▪	51	3
20	DC4	▪	52	4
21	NAK	▪	53	5
22	SYN	▪	54	6
23	ETB	▪	55	7
24	CAN	▪	56	8
25	EM	▪	57	9
26	SUB	▪	58	:
27	ESC**	▪	59	;
28	FS	▪	60	<
29	GS	▪	61	=
30	RS	▪	62	>
31	US	▪	63	?

*THE ASCII CHARACTER DIFFERS FROM THE DISPLAYED CHARACTER FOR CODES 00 - 31 ONLY.

2280068

Table B-1. ASCII Character/Graphic Character Codes (Continued)

I CODE	U.S. FRANCE JAPAN DISPLAYED CHARACTER	U.K. GERMAN DISPLAYED CHARACTER	DENMARK NORWAY DISPLAYED CHARACTER	SWEDEN FINLAND DISPLAYED CHARACTER
64	@	@	@	É (SWEDISH)
65	A	A	A	A
66	B	B	B	B
67	C	C	C	C
68	D	D	D	D
69	E	E	E	E
70	F	F	F	F
71	G	G	G	G
72	H	H	H	H
73	I	I	I	I
74	J	J	J	J
75	K	K	K	K
76	L	L	L	L
77	M	M	M	M
78	N	N	N	N
79	O	O	O	O
80	P	P	P	P
81	Q	Q	Q	Q
82	R	R	R	R
83	S	S	S	S
84	T	T	T	T
85	U	U	U	U
86	V	V	V	V
87	W	W	W	W
88	X	X	X	X
89	Y	Y	Y	Y
90	Z	Z	Z	Z
91	[Ä	Æ	Ä
92	\	Ö	ϕ	Ö
93]	Ü	Å	Å
94	^	^	^	Û (SWEDISH)
95	-	-	--	-

228 0066

Table B-1. ASCII Character/Graphic Character Codes (Continued)

I CODE	U.S. U.K. FRANCE JAPAN DISPLAYED CHARACTER	GERMAN DISPLAYED CHARACTER	DENMARK NORWAY DISPLAYED CHARACTER	SWEDEN FINLAND DISPLAYED CHARACTER
96	\	\	\	é (SWEDISH)
97	a	a	a	a
98	b	b	b	b
99	c	c	c	c
100	d	d	d	d
101	e	e	e	e
102	f	f	f	f
103	g	g	g	g
104	h	h	h	h
105	i	i	i	i
106	j	j	j	j
107	k	k	k	k
108	l	l	l	l
109	m	m	m	m
110	n	n	n	n
111	o	o	o	o
112	p	p	p	p
113	q	q	q	q
114	r	r	r	r
115	s	s	s	s
116	t	t	t	t
117	u	u	u	u
118	v	v	v	v
119	w	w	w	w
120	x	x	x	x
121	y	y	y	y
122	z	z	z	z
123	{	ä	æ	å
124		ö	ø	ö
125	}	ü	å	å
126	~	ß	~	ü
127	DEL	DEL	DEL	DEL

2280065

Table B-1. ASCII Character/Graphic Character Codes (Continued)

I CODE	JAPANESE DISPLAYED CHARACTER	I CODE	JAPANESE DISPLAYED CHARACTER	I CODE	JAPANESE DISPLAYED CHARACTER	I CODE	JAPANESE DISPLAYED CHARACTER
128	-	160	NONE	192	タ TA	224	?
129	■	161	〇	193	チ CHI	225	!
130	■	162	□	194	ツ TSU	226	"
131	■	163	∩	195	テ TE	227	#
132	■	164	∪	196	ト	228	\$
133	■	165	・	197	ナ NA	229	%
134	■	166	ヲ	198	ニ NI	230	&
135	■	167	ア A	199	ヌ NU	231	'
136	■	168	イ I	200	ネ NE	232	(
137	■	169	ウ U	201	ノ NO	233)
138	■	170	エ E	202	ハ HA	234	*
139	■	171	オ O	203	ヒ HI	235	+
140	■	172	ヤ YA	204	フ FU	236	,
141	■	173	ユ YU	205	ヘ HE	237	-
142	■	174	ヨ YO	206	ホ HO	238	.
143	■	175	ツ TSU	207	マ MA	239	/
144	■	176	ー	208	ミ MI	240	0
145	■	177	ア A	209	ム MU	241	1
146	■	178	イ I	210	メ ME	242	2
147	■	179	ウ U	211	モ MO	243	3
148	■	180	エ E	212	ヤ YA	244	4
149	■	181	オ O	213	ユ YU	245	5
150	■	182	カ KA	214	ヨ YO	246	6
151	■	183	キ KI	215	ラ RA	247	7
152	■	184	ク KU	216	リ RI	248	8
153	■	185	ケ KE	217	ル RU	249	9
154	■	186	コ KO	218	レ RE	250	:
155	■	187	サ SA	219	ロ RO	251	;
156	■	188	シ SHI	220	ワ WA	252	'
157	■	189	ス SU	221	ン N	253	>
158	■	190	セ SE	222	〃	254	>
159	■	191	ソ SO	223	〇	255	DEL

2280064

Appendix C

BASIC Reserved Word List

The following are reserved words for TI BASIC.

ABS	ESUB	OLD
ACCEPT	EXP	ON
ALL	FIXED	OPEN
AND	FNEND	OPTION
APPEND	FOR	OR
ASC	FREESPACE	OUTPUT
ASSIGN	FTYPE	PERMANENT
AT	GO	POS
ATN	GOSUB	PRINT
BASE	GOTO	PUNCTUATION
BELL	IF	RANDOMIZE
BREAK	IMAGE	READ
BRK	INKEY	REAL
BRKPNT	INKEY\$	REC
BYE	INPUT	RELATIVE
CALL	INT	REM
CHR\$	INTEGER	REN
CLOSE	INTERNAL	RENAME
COS	KEY	REPRINT
DAT\$	KEYED	RES
DATA	LEN	RESEQUENCE
DECIMAL	LET	RESTORE
DEF	LIBRARY	RETURN
DEL	LIS	RND
DELETE	LIST	RPT\$
DIM	LOCK	RUN
DISPLAY	LOG	SAV
DUP	MAIN	SAVE
EDI	MER	SCRATCH
EDIT	MERGE	SEG\$
ELSE	NEW	SEQUENTIAL
END	NEXT	SGN
EOF	NOT	SIN
ERASE	NUM	SIZE
ERR	NUMBER	SPAN
ERROR	NUMERIC	SQR

BASIC Reserved Word List

STEP
STOP
STR\$
SUB
SUBEND
SUBEXIT
TAB
TAN
TEMPORARY

THEN
TIME\$
TO
TRA
TRACE
UNB
UNBRKPT
UNLOCK
UNT

UNTRACE
UPD
UPDATE
UPRC\$
USING
VAL
VARIABLE

Appendix D

Error Messages and Codes

D.1 INTRODUCTION

This section lists and describes the BASIC error messages. The format for BASIC error messages is as follows:

ERROR # error__number IN line__number

WARNING # error__number IN line__number

STOPPED IN line__number

The error__number parameter specifies a particular error encountered. The text following the number in the table describes conditions that cause the error. When IN line__number is included, it specifies the line number of the statement being evaluated when the error was detected. Usually, this line contains the error. If the error message results from a statement executed in immediate execution mode, the IN line__number field is not displayed.

When an error message is encountered, the BASIC system does not attempt to recover. If no ON ERROR statement is in effect, the system displays the error message and returns to command mode. At this point, use the LIST command to list the indicated line number and then read the indicated error description in Table D-1. If appropriate, examine the relevant variables by entering the variable names and using the calculate function key.

When a warning message occurs, the BASIC system performs a recovery procedure. If no ON ERROR statement is in effect, the system displays the error message and continues execution. Table D-1 describes the recovery procedure for each warning message.

A STOPPED message appears if you press the breakpoint, break key, or step key. The program remains stopped until you press the resume execution function key.

The following is an example of a BASIC error message.

```
ERROR #6 IN 125
```

where:

ERROR indicates that there is no recovery procedure for the encountered error.

#6 indicates that the description of the error follows error number 6.

IN 125 indicates that the error was detected while executing a statement in line number 125.

If this error message is displayed and no ON ERROR statement is in effect, the system returns to command mode. Executing the command LIST 125 displays a statement that has one or more references to the log function. By entering the arguments to each of the log function references and pressing the calculate function key, you can determine the negative argument.

When an ON ERROR statement is in effect, an error message is not printed unless a RETURN PRINT statement is executed before the next error or warning is encountered. A program can easily differentiate between errors and warnings because warning error codes are returned as the negative of the error number when the ERR function is evaluated.

BASIC can detect operating system I/O errors as well as BASIC program errors by using the ON ERROR statement. If error trapping is not in effect at the time of an I/O error, the system returns an error message consisting of the ERR value and the equivalent error code (hexadecimal). The following example shows error trapping.

EXAMPLE

ERROR #1183 (OS > B7) IN LINE 120

The error code indicates that an attempt was made to access a locked record. Similar error messages appear on the other TI operating systems that support BASIC. To look up an error message, refer to your operating system error message manual.

Table D-1. BASIC Error Messages

Error Number	Error Message/Description
1	<p>BREAKPOINT The break execution key was pressed or the program encountered a breakpoint at the indicated line. Press the resume execution key to resume execution of the program.</p>
2	<p>MISSING OR MISTYPED NUMBER BASIC expected a numeric value, but the data was either missing or of the wrong type. An example of this is VAL("A4"), where VAL requires a character string representing a numeric value.</p>
5	<p>DIVISION BY ZERO The indicated statement specifies division by zero. Execution continues, using the largest number representable. Examine the current values of divisors in the statement to determine which one is zero, and modify the program appropriately.</p>
6	<p>LOG OF A NON-POSITIVE NUMBER The LOG function in the indicated statement has a negative or zero argument.</p>
7	<p>NEGATIVE NUMBER TO NON-INTEGGER POWER The statement specifies that a negative value be raised to a noninteger power. Since this would result in a complex value, an error occurs.</p>
8	<p>SQUARE ROOT OF NEGATIVE NUMBER The square root function, SQR, has a negative argument.</p>
10	<p>ARITHMETIC OVERFLOW A calculation produced a number that is greater than or equal to 1.E128. It has been replaced with the largest number the system can handle, and execution continues.</p>
11	<p>INTEGER OVERFLOW An attempt to calculate an integer value greater than 32,767 was encountered, and execution stopped. You can change the operand to type REAL to alleviate the problem.</p>
12	<p>DECIMAL OVERFLOW This statement attempts to store a decimal value that exceeds the limits imposed by scaling. This condition also results when a DECIMAL statement has a scale size greater than 15.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
14	<p>ERROR IN SYNTAX The statement indicated has an error in its syntax. The error is usually the result of a misspelled or misused keyword, an incorrectly placed or missing separator, or an improperly formatted constant. Refer to the discussion of the appropriate statement for the correct syntax. This can also indicate an error in the data being read by the statement (for example, data elements not separated by commas).</p>
15	<p>ERROR IN DIMENSION In the indicated statement, the syntax is in error or the size of a specified virtual array exceeds 65,536 elements.</p>
16	<p>STATEMENTS BETWEEN SUBPROGRAMS The program contains statements after a SUBEND and before a following SUB statement. Remove these statements, since they cannot be executed.</p>
17	<p>NONTERMINATED QUOTED STRING The statement contains a quote (") that is not paired (for example, CH\$ = "ABC). Add the missing quote to the statement.</p>
18	<p>ILLEGAL CHARACTER DETECTED The program in memory contains a character that is illegal in a BASIC program. Either the program was generated outside the BASIC system or a flaw has occurred in the storage media.</p>
19	<p>ILLEGAL KEYWORD FOR SAVE WITH LOCK An attempt was made to save a program with the lock option specified, which contains one of the following keywords: MERGE, BRKPNT, UNBRKPNT, TRACE, UNTRACE. You should remove these keywords from programs before saving them with LOCK.</p>
20	<p>KEYWORD OUT OF CONTEXT OR SYNTAX ERROR The statement contained a keyword where one was not expected. This error can result from using keywords as variable names, as in ON THEN GOTO 40,50. If this is not the case, check the syntax.</p>
22	<p>SYMBOL NOT FOUND OR CREATABLE The statement contains a symbol that was never defined in the program. A symbol becomes defined when it is the object of a LET, READ, INPUT, ACCEPT, FOR, or CALL statement. Add a statement that establishes the initial value of the variable. This message also results if a program file is improperly specified.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
24	<p>DATA TYPE MISMATCH The statement contains a value that does not correspond to the type required. For example, a numeric expression contains a string constant or variable, a string expression contains a numeric constant or variable, a statement references a virtual array with a different type of variable from the one with which it was created, or a statement references a subroutine or function with arguments that do not correspond to the type with which it was defined.</p>
25	<p>ERROR IN DECLARATION This declarative statement is improperly positioned or formed. For example, an OPTION BASE statement has neither a 0 nor a 1 argument, an ASSIGN statement includes an improperly named variable, or a MERGE statement is not at the beginning of a program.</p>
28	<p>MULTIPLY DECLARED OR USED VARIABLE A variable appeared in more than one declaration. (For example, INTEGER X::REAL X was referenced with a different number of dimensions from that with which it was defined or was used as a function and as a variable within the same program.) In some cases, this error can occur when saving a program with LOCK specified, even though the program executed with no error when not locked. In these cases, an array has been referenced in a line preceding the line in which it was defined (using a DEF, INTEGER, REAL or DECIMAL statement). The definition should be moved to a line preceding the first reference to the array.</p>
29	<p>ILLEGAL FOR VARIABLE The statement specified an array element or a string variable as the index of a FOR statement; for example, FOR V(1)= 1 TO 5.</p>
30	<p>CANNOT ASSIGN TO FUNCTION NAME The statement results in an attempt to read into a function name while in command mode.</p>
31	<p>NO SUCH VIRTUAL ARRAY The statement references a relative record file that was not created with a virtual array. This also results when a command references a virtual array element.</p>
32	<p>UNASSIGNED VIRTUAL ARRAY The statement references an element of a virtual array for which the ASSIGN statement has not been executed, or the virtual array has been closed.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
33	<p>ILLEGAL INPUT OR ACCEPT VARIABLE An INPUT or ACCEPT statement executed in command mode has the name of a function as an input variable.</p>
36	<p>IMAGE FIELD HAS OVER 14 SIGNIFICANT DIGITS The specified image has more than 14 digits of precision specified for a single field. Reduce the number of #s in the image.</p>
37	<p>NO DATA FIELDS IN IMAGE The specified image contains no data conversion fields, but data is provided in the PRINT USING statement. Either add conversion fields to the image or eliminate the variable list from the PRINT USING statement.</p>
39	<p>OUT OF MEMORY First attempt to increase the workspace. The present level of usage will determine whether and to what extent an increase is possible. Or, if this error occurs immediately after RUN, the combination of program space and variables is too large. Reduce the dimensions of arrays, change large arrays to virtual arrays, or break the program into sections that link by means of the RUN statement and communicate by means of files. If the error occurs later in execution, the cumulative length of the string variables is too large. Recover the space occupied by inactive strings by setting the string to empty (for example, A\$ = "").</p>
40	<p>STACK OVERFLOW The operation stack has exceeded its limits. This can be the result of too many GOSUBs without a RETURN being executed, too many recursive function calls, too many CALLs without a SUBEND or SUBEXIT being executed, too many incomplete FOR-NEXT loops, too many ON ERROR statements being executed without a RETURN, or a complex expression.</p>
42	<p>FNEND NOT INSIDE A FUNCTION CALL GOTO in a program branched into the definition of a function. You can execute a function only by referencing the function name in an expression. Correct the flow of the program. Use the TRACE statement to help determine where the flow requires correction.</p>
43	<p>NEXT WITHOUT FOR GOTO has transferred control into the range of a FOR loop, and a NEXT statement is encountered without the associated FOR statement being executed. Executing with the TRACE feature usually locates the point at which the flow was disrupted.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
45	FOR/NEXT/DEF/FNEND INSIDE AN IF An IF statement contains a FOR, NEXT, DEF, or FNEND. Change the IF to branch to statements that contain the required statement types.
46	TOO MANY ELSESES The statement contains more ELSEs than IFs. Restructure the statement so that this does not occur.
47	SUBEND NOT INSIDE A CALL The GOTO in a program has transferred control into a subroutine and a SUBEND is encountered. You can enter subroutines only by executing a CALL. Running the program using TRACE should point out where the subroutine was improperly entered.
48	RECURSIVE SUBROUTINE CALL The program has called this subroutine a second time before its SUBEND or SUBEXIT has been executed. Change the flow of the program so that the subroutine will not be called again, or call another subroutine that causes the first subroutine to be called.
49	MISSING SUBEND The subroutine specified is not terminated by a SUBEND. Append a SUBEND statement to the end of the BASIC program.
50	IMPROPER NESTING OF FORS AND DEFS The program encountered a second DEF statement before encountering an FNEND statement for the DEF currently active, the program encountered an FNEND between a FOR and its corresponding NEXT, or the statements bound by two FOR/NEXT pairs overlapped instead of being nested or separate. Rearrange the statements involved.
51	RETURN WITHOUT GOSUB OR ON ERROR The program encountered a RETURN statement for which there was no corresponding GOSUB or ON ERROR. Possibly, a GOTO rather than an intended GOSUB was executed earlier in the program.
52	ILLEGAL COMMAND IN AN EXTERNAL SUBROUTINE You cannot use top-level commands within an external subroutine, or one external subroutine cannot call another within a program.
54	STRING TOO LONG — TRUNCATED The concatenation operation resulted in a string over 255 characters long. The excess was lost, but execution continued.

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
55	<p>INPUT FIELD EXCEEDS AVAILABLE LINE LENGTH The specified input prompt in an INPUT or ACCEPT statement exceeds the number of characters available on the line. Modify the program so that the string expression does not exceed the acceptable length. Alternatively, if more characters are required, issue a DISPLAY AT with the prompt message followed by an ACCEPT AT with no prompt.</p>
57	<p>SUBSCRIPT OUT OF RANGE A subscript reference exceeded the range of its dimensions; for example, Q(12) = 5, where the dimensions of Q are declared by reference or specification to be less than 12. Specify the desired maximum subscripts explicitly in a DIM statement.</p>
58	<p>WRONG NUMBER OF SUBSCRIPTS The statement referenced an array with fewer or more subscripts than were initially specified for the array. Each dimension must have only one corresponding subscript in each reference to the array.</p>
60	<p>LINE NOT FOUND A transfer statement (for example, GOTO) has as its destination a nonexistent statement. Either add the specified statement or correct the transfer statement.</p>
61	<p>BAD LINE SPECIFICATION You entered incorrectly a command that uses a line number increment or line range, executed a RESEQUENCE command containing an argument that is not a number or that results in a line number greater than 32,759, or included a USING clause that referenced a nonexistent line.</p>
63	<p>INDEX OUT OF RANGE The argument of an ON . . . GOTO statement is larger than the number of line numbers given.</p>
64	<p>STEP/LINE NUMBER/RECORD SIZE CANNOT BE ZERO The statement specified a step size in a FOR statement, a record size in an OPEN statement, or a line number of 0.</p>
65	<p>ATTEMPT TO ACCESS PROTECTED PROGRAM The statement attempts to list, save, or alter a protected program.</p>
66	<p>DIRECT COMMAND TERMINATED You executed a CALL or GOSUB statement in command mode and pressed the continue function key, creating an ambiguous execution sequence.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
67	<p>CANNOT CONTINUE You are attempting to continue from a fatal error or to begin program execution by pressing the resume execution function key without having entered the RUN command. Correct any errors and enter the RUN command.</p>
68	<p>UNACCEPTABLE RESEQUENCE SPECIFICATIONS The specifications for an RES are not acceptable. Either the values are out of acceptable line number range or executing the RES results in changing the relative position of statements in memory (for example, RES 30000,10000).</p>
69	<p>COMMAND ILLEGAL IN PROGRAM The command used in the indicated line can be used only in top-level commands (for example, RES). Delete the statement.</p>
70	<p>CAN ONLY BE USED IN A PROGRAM The command just entered (for example, DIM) can be used only in a program.</p>
71	<p>NO LINE NUMBER During the loading of a program, BASIC encountered a line without a line number.</p>
74	<p>BAD ARGUMENT TO INTRINSIC FUNCTION/CALL One of the arguments used in referencing a function or subroutine does not have the attributes required by that procedure (e.g., negative string function indices).</p>
75	<p>WRONG NUMBER OF ARGUMENTS TO USER FUNCTION OR SYSTEM SUBROUTINE The statement references a user-defined function or system subroutine with a different number of arguments than was specified in the function or subroutine definition.</p>
76	<p>CAN'T CALL BY REFERENCE An attempt was made to pass an array element in place of an array. Either remove the subscript from the call or transmit it as a separate parameter. Some other incidents of variable attribute mismatches can also cause this error message.</p>
77	<p>CANNOT PASS VIRTUAL ELEMENT BY REFERENCE The program is attempting to pass a virtual array element as an argument. Enclose the argument in parentheses if it is a single element.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
78	<p>NO PROGRAM TO SAVE A SAVE command was entered but no program is in memory. Verify this by entering the LIST command. Reenter the program.</p>
79	<p>INVALID USE OF FUNCTION A user-defined function invocation may not be a parameter in a CALL to an external subprogram.</p>
80	<p>TOO MUCH INPUT DATA You have provided more data in an input line than is requested. Reenter the entire corrected line.</p>
81	<p>TOO MANY VARIABLES IN CALL OR INPUT More variables are specified in a CALL or INPUT statement than are specified in the corresponding SUB statement or input record.</p>
83	<p>TOO LITTLE INPUT DATA You have supplied less data than was requested. Reenter the entire line.</p>
84	<p>OUT OF DATA READ statements have exhausted the data provided in DATA statements. Either restore the data pointer or add more DATA statements and then run the program.</p>
87	<p>BASIC UNIT NUMBER OUT OF RANGE You attempted to specify a BASIC unit number outside the legal range (1 to 255); e.g., you attempted to close unit number 0.</p>
89	<p>BAD UNIT NUMBER SPECIFICATION The program is attempting to perform I/O using a file with an illegal logical unit number. Acceptable unit numbers are 1 to 255. This error generally occurs in an OPEN statement.</p>
90	<p>UNIT NUMBER ALREADY ASSIGNED The program is attempting to open a file using a logical unit number that is already assigned to another file. Close the first file before attempting to open the second, or specify another logical unit number.</p>
91	<p>ILLEGAL OPERATION ON UNOPENED FILE An INPUT, PRINT, CLOSE, or RESTORE statement is referencing a logical unit number that is not associated with a currently open file. Verify that the unit number used in the referenced statement is the same one specified in the associated OPEN statement.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
92	<p>ILLEGAL OPERATION IN BACKGROUND MODE (DX BASIC ONLY) The program running in background mode has attempted to direct data to the terminal through a PRINT, ACCEPT, INPUT, or DISPLAY statement.</p>
96	<p>FILE UNEXPECTEDLY CLOSED During the evaluation of an input or output statement, the program executed a function that closed the referenced file. Close the file only after the statement has been completed.</p>
101	<p>ILLEGAL READ/WRITE TRANSITION You attempted to proceed from a read operation to a write operation or from a write operation to a read operation without an intervening CLOSE or RESTORE. Close the file and reopen it, or restore the file to the beginning. This error occurs only with sequential files.</p>
104	<p>CANNOT APPEND TO RELATIVE FILE The program is attempting to open a relative record file, specifying the APPEND option. This option is valid only with sequential files.</p>
105	<p>DESIRED I/O VIOLATES OPEN MODE The indicated I/O operation conflicts with the mode specified when the file was opened. A PRINT statement referenced a file not opened OUTPUT, UPDATE, or APPEND. An INPUT or ACCEPT statement referenced a file not opened INPUT or UPDATE. An attempt was made to RESTORE a device. A REC clause was used with a sequential file.</p>
106	<p>INVALID OPEN ATTRIBUTE MIX The program has executed an OPEN statement with an illegal combination of attributes (for example, RELATIVE, APPEND). Refer to the OPEN statement discussion for legal combinations.</p>
107	<p>INCORRECT FILE TYPE The program is attempting to open a file with specifications different from those used when the file was created. This error occurs if you attempt to open a relative record file with the SEQUENTIAL specification implied or listed in the OPEN statement, or vice versa.</p>
108	<p>IMPROPER LENGTH SPECIFICATION The OPEN statement specified an illegal record length of less than 2 or greater than 32,544 bytes. Also, if the record length is fixed and the length specified is not equal to the length specified when the file was created, this error results.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
109	<p>KEY FILE ERROR The indicated key number is beyond the number of keys specified when the file was created.</p>
115	<p>BAD VALUE IN A REC CLAUSE This error appears when you are performing file I/O if you supply a value greater than five trillion or supply a negative value.</p>
128	<p>ERROR IN DISK OPERATION BASIC detected an error during a disk operation. This error message can indicate a bad area on the disk or a hardware failure. Retry the operation.</p>
135	<p>UNKNOWN SUBPROGRAM The CALL statement attempted to transfer program control to a subprogram that could not be located. Refer to the discussion of the CALL statement for the location and calling sequence of subprograms.</p>
138	<p>DUPLICATE FILE NAMES The user or program is attempting to create or rename a file by using the name of an already existing file.</p>
145	<p>INVALID REAL NUMBER The program is attempting to input data from a relative record file but the data is inconsistent with the type of variables in the input list.</p>
148	<p>END-OF-FILE You attempted to read past the last record in a file.</p>

Table D-1. BASIC Error Messages (Continued)

Error Number	Error Message/Description
160	<p>FATAL INTERNAL ERROR Either the computer malfunctioned or an unanticipated sequence of events occurred. Please report this occurrence to your Texas Instruments representative. This error is also generated if you attempt to execute a program that was not saved in the LIST format with an earlier system.</p>
164	<p>BAD LITERAL STRING You used a non-ASCII character inside a literal string.</p>
171	<p>INTERNAL ERROR This error should not occur. It indicates an error in TI BASIC.</p>
172	<p>SYSTEM ERROR A fatal error occurred in BASIC. BASIC terminates and returns control to the operating system.</p>
173	<p>WORK FILE ERROR A fatal error occurred in attempting to create or access the work file. BASIC terminates and returns control to the operating system. Make sure that the work file specified has adequate available disk space and directory entries, and that the work file volume is not write-protected.</p>
174	<p>OVERLAY FILE I/O ERROR A fatal error; if BASIC is installed correctly, this is a system error.</p>
175	<p>OVERLAY MANAGER STACK OVERFLOW A system error.</p>
1000	<p>OPERATING SYSTEM ERROR NUMBER Error numbers greater than 1000 refer to operating system error codes. The number of the error is equal to 1000 plus the decimal equivalent of operating system error code. For example, the error number for DX10 error > B7 (decimal 183) is ERROR #1183. For details, refer to the error messages and codes manual that accompanies your operating system.</p>

D.2 SUBROUTINE ERROR MESSAGES

Table D-2 contains a description by subroutine of error codes returned by the KFP and sort subroutines. This table contains both error codes returned by the subroutine error status argument (STAT) and errors that may occur in the BASIC CALL statement.

Table D-2. Subroutine Error Messages

Error Code	Subroutine	Meaning
0	All*	Subroutine executed successfully; no error detected.
39	All*	Subroutine required more freespace than was available. This error is an error in the BASIC CALL statement.
74	All*	One of the arguments used in calling the subroutine does not have the attributes required by that subroutine. This may be the result of improper data type (e.g., supplying a REAL value when an INTEGER value is required), or improper argument initialization (e.g., supplying a string whose length is less than that required by the subroutine or an integer whose value is not within a required range). This error is an error in the BASIC CALL statement.
75	All*	Wrong number of arguments used when calling the subroutine. This error is an error in the BASIC CALL statement.
80	All*	The user has provided more data in an input line than is requested. Reenter the entire corrected line.
86	All*	A name has been specified that is not a properly formed or legal file name, device name, or volume name.
97	All*	Hardware protection violation. The user has attempted to write to a disk protected from output.
107	All*	Invalid file type. The user has attempted to sort a sequential file.
110	All*	An attempt has been made to access a file in a manner contrary to its protection status (e.g., delete a write or delete protected file).
128	All*	BASIC detected an error during a disk operation, which can indicate a bad area on the disk or a hardware failure. Retry the operation.
129	All*	An attempt has been made to perform I/O with a disk drive without a properly installed disk.
134	All*	The user attempted to access a file that is not on the specified disk drive or a device that is not in the system configuration.

Table D-2. Subroutine Error Messages (Continued)

Error Code	Subroutine	Meaning
148	All*	The user has attempted to read beyond the last record of a file.
149	All*	The program is attempting to write to a file beyond the space allocated when the file was created. To write more data, copy the file to a file with a larger allocation. This error occurs when the user attempts to save a file in list format with insufficient space on the disk.
204	All*	An attempt has been made to perform an operation that is illegal for the specified device (e.g., input from an 810 printer).
275	SORT* MERGE*	Too many key fields (more than ten).
276	SORT* MERGE*	Key fields too long. The combined length of all of the key fields is larger than the size parameter.
278	SORT*	Too many records to sort. See the subroutine sorting limits description in Section 12.
279	SORT* MERGE*	Too many variables in input record (more than 100 fields).
280	SORT* MERGE*	Not enough memory.
281	SORT* MERGE*	Error in key field description.
350	KFCLOS KFOELR KFGET KFINIT KFOPEN KFPUT KFREAD KFWRIT	The KFP data base buffer is not configured into the system. Run the BLDBUF utility, then load the buffer file created by BLDBUF (see Section 12).
351	KFCLOS KFDEL KFGET KFOPEN KFPUT KFREAD KFWRIT	An attempt has been made to execute a KFP subroutine prior to initializing the data base buffer. Run the KFINIT subroutine to initialize the buffer.

Table D-2. Subroutine Error Messages (Continued)

Error Code	Subroutine	Meaning
352	KFCLOS KFDEL KFGET KFPUT KFREAD KFWRITE	A KFP subroutine has attempted to access a file using an invalid file unit number.
353	KFCLOS KFDEL KFGET KFPUT KFREAD KFWRITE	A KFP subroutine has attempted to access an unopened file.
354	KFCREA	The maximum key size argument (MAXKEY) is larger than $((\text{page size} - 6)/2) - 2$.
355	KFINIT	The subroutine has been called with a keyed file currently open.
356	KFINIT	The configured KFP data base buffer size is too small for the parameters specified.
357	KFINIT	One of the input values is too small or too large.
358	KFINIT	The maximum record size supplied is greater than $(\text{page size} - 6)/2$.
359	KFINIT	The page or record size supplied has an odd value.
360	KFINIT	The maximum number of fields supplied is greater than $(\text{record size})/2$.
361	KFINIT	The maximum number of files supplied is greater than $(\text{number of pages})/2$.
362	KFOPEN	The maximum number of open files has been exceeded.
363	KFWRITE	The record buffer is empty during execution of a write operation. The KFPUT subroutine may not have been executed before the write.
364	KFOPEN	The user has attempted to open a file not created as a KFP file.
365	KFOPEN	The physical record size of this keyed file exceeds the maximum page size.
366	KFCREA KFGET KFPUT	The field number argument is out of range.
367	KFGET KFPUT	The supplied data arguments do not match the record format template used during file creation.

Table D-2. Subroutine Error Messages (Continued)

Error Code	Subroutine	Meaning
368	KFGET KFPUT	The user has attempted to access the record buffer prior to initializing it.
369	KFPUT	The record buffer is too small to initialize for this keyed file (too many data elements).
370	KFPUT	The user has attempted to overflow the record buffer during the put operation. No arguments have been placed in the buffer.
371	KFCREA	A syntax error has occurred in the record field format template or in the key field template, or the maximum number of fields in the template has been exceeded (100 for record field template; 10 for key field template). This error can also occur if the number of key fields exceeds the number of fields defined.
372	KFWRIT	The user attempted to rewrite an existing record within the keyed file without selecting the replace option.
373	KFDEL KFREAD KFWRIT	A page search error has occurred (e.g., the specified record cannot be found). If this error occurs, file integrity has probably been lost. Close and then reopen the keyed file and retry the operation. If the error persists, restore the file from the back-up media.
374	KFDEL KFREAD	The supplied key argument does not match the key type specified during file creation.
375	KFREAD	An invalid read opcode was used.
376	KFREAD	The keyed file record just read does not fit into the record buffer. The contents of the buffer is unchanged.
377	KFDEL KFREAD	An attempt has been made to delete or read (equal) by a key that does not exist in the file.
378	KFWRIT	The pending write operation may exceed the file allocation. Data is not written to the file. Enlarge the file to accommodate more records.
379	KFREAD	The read operation attempted to read beyond the end or beginning of the file.
380	KFWRIT KFREAD KFDEL	The concatenated length of the keyed field(s) exceeds the maximum composite key size established by KFCREA.

Note:

* These errors may have a prefix indicating that the error occurred on an input file (1XXX), output file (2XXX), or work file (3XXX), where XXX indicates the error number. This notation occurs only with errors returned by the sort subroutine.

Appendix E

BASIC System Differences

E.1 INTRODUCTION

TI BASIC includes the following:

- DX10 BASIC — Operates on members of the TI family of computers that use the DX10 operating system
- DNOS BASIC — Operates on members of the TI family of computers that use the DNOS operating system
- DX10 Micro BASIC — Operates on the Business System 200 computers that use the DX10 Micro (DXM) operating system

The following paragraphs identify system differences for users who are familiar with TI BASIC systems and for those who wish to become familiar with them for future use.

E.2 KEYBOARD AND FUNCTION KEYS

The keyboard and function keys vary according to the hardware configuration. Appendix I lists the functions supported by each terminal device and specifies which keys control each function.

E.3 FILE CHARACTERISTICS

The following paragraphs describe the significant file characteristics that differ among the TI BASIC systems.

E.3.1 Temporary Files

DXM does not support temporary files; DX10 and DNOS do. If you attempt to create a temporary file on a system using the DXM operating system, an error message appears.

E.3.2 DXM File Size Restrictions

The size of buffers on the DXM operating system limits the size of files you can use. Any attempt to open a file exceeding the maximum record length results in an overflow of the system table area. The preferred physical record length for files on the Winchester disk is 256; the preferred physical record size for files on an 8-inch diskette is 288 bytes.

E.3.3 Pathnames

The pathnames that you can use in BASIC statements and commands are completely operating-system dependent.

E.3.3.1 DX10 and DNOS Pathnames. In DX10 and DNOS, a pathname consists of a volume name, the directory names leading to a file, and the file name. The names in the pathname are separated by periods. The volume name need not be included if the file or directories leading to the file reside on the system disk. The number of directories leading to the filename are not restricted except that the total number of characters in the pathname must not exceed 48. The first character of each name must be alphabetic. The following is an example of a DX10 pathname:

MYVOL.MYDIR.OBJ.PB04

E.3.3.2 DXM Pathnames. DXM allows a maximum of three levels of structure in a file name: the volume name, an optional user-created directory name, and a file name. For example, a file name can be DXM.FILENAME or can be DXM.DIRECTRY.FILENAME. There are a maximum of 26 letters in a valid DXM pathname (including the periods).

CAUTION

You can begin a file pathname with a device rather than a volume name, but we advise against it. If you do begin a pathname with a device name, be sure that the correct volume is in the requested drive.

DXM does not have the built-in precaution that the DX10 and DNOS systems have, of requiring you to install a volume by entering the Install Volume (IV) command. DXM automatically installs all volumes physically in the system. If you send output to DS01.CONTRL and you meant to send it to USERVOL.CONTRL that is actually in drive DS02, output will go automatically to DS01. If you have another file by the name CONTRL on that disk, you may get an error message or you may replace the original file.

E.4 DXM PROGRAM SIZE RESTRICTIONS

The total amount of memory available on the DXM system is 64K bytes. The maximum size of a BASIC task that can be run on a DXM system is limited by the size of the resident part of the operating system and the size of the DXM BASIC interpreter.

E.5 INTERTASK SUPPORT

DX10 and DNOS support intertask communications; DXM does not. BASIC application programs that interface to another task through intertask communications or via shared procedures are not supported on DXM because the operating system does not have multitasking capabilities. For this reason, you cannot execute programs that have calls to other programs such as Sort/Merge on a DXM system.

E.6 THE BELL

The bell will not sound on an S300 computer or on a 940 terminal.

E.7 DNOS SPOOLER DEVICES

Under DNOS, you cannot open a printer that is defined as a spooler device. If you attempt to do so, a >9D error message appears.

Appendix F

Logical Operators with Integer Operands

F.1 INTRODUCTION

Whenever a relational operator is evaluated, the result is true or false. The numeric values of true and false resulting from these operations are -1 and 0, respectively. These values are stored as integers. Thus, the logical value false is stored as hexadecimal 0000, and the logical value true is stored as hexadecimal FFFF.

Logical operators use the full 16-bit representations for true and false. As long as 0 and -1 are the only values used, the result of any AND, OR, or NOT operation is always 0 or -1. Thus, the logical operators provide the logic value operations described in Section 4. However, BASIC also allows logic operations to apply to any numeric value. If a noninteger argument is applied to a logical operator, it is first converted to an integer. Although the results no longer necessarily equal logic values, they are still valid integers and the operations are frequently useful. Since hexadecimal numbers cannot be directly represented in BASIC, the corresponding integer values must be used.

F.2 LOGICAL OR OPERATOR

The following indicates the results of applying the logical OR operator to bits X and Y:

Value of X	Value of Y	Value of X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

As indicated by the table, the result of two bits being ORed is equal to 1 if either or both bits equal 1. The following example ORs two 16-bit words.

WORD #1:	0 0 1 0 0 1 1 0 0 0 0 1 0 0 1 1
WORD #2:	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
WORD #3:	0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 1

This example demonstrates how an OR operator can set bits to 1. The leftmost byte of word 1 was ORed with a zero byte; the leftmost byte of the resulting word remained the same as that of word 1. However, the rightmost byte of word 1 was ORed with a byte of all ones; the rightmost byte of the result was then equal to all ones, regardless of the contents of word 1.

To perform the OR operation in the preceding example, the system must convert the values of words 1 and 2 to base-10 integers. Using base-10, word 1 equals 9747 and word 2 equals 255. Executing the following command produces the result 9983, which is the base 10 equivalent of word 3:

```
.PRINT 9747 OR 255
```

F.3 LOGICAL AND OPERATOR

The following indicates the results of applying the logical AND operator to bits X and Y:

Value of X	Value of Y	Value of X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

As indicated by the table, the result of two bits being ANDed is equal to 1 only if both bits are equal to 1. The following example ANDs two 16-bit words.

```
WORD #1: 0 0 1 0 0 1 1 0 0 0 0 1 0 0 1 1
WORD #2: 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
WORD #3: 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1
```

This example demonstrates how an AND operation can set bits to 0. The leftmost byte of word 1 was ANDed with a zero byte; the leftmost byte of the result was then equal to all zeros, regardless of the contents of word 1. However, the rightmost byte of word 1 was ANDed with a byte of all ones; the rightmost byte of the resulting word remained the same as that of word 1.

For BASIC to perform the AND operation in the above example, the values of words 1 and 2 must first be converted to base-10 integers. Executing the following command produces the result 19, which is the base-10 equivalent of word 3:

```
.PRINT 9747 AND 255
```

F.4 LOGICAL NOT OPERATOR

The NOT operator takes only one operand. The following indicates the results of applying the logical NOT operator to bit X:

Value of X	Value of NOT X
0	1
1	0

As indicated by the table, applying the NOT operator to a bit changes a 0 to a 1 and vice versa. The following example executes the NOT operation on a full 16-bit word:

```

WORD #1:    0 0 1 0 0 1 1 0 0 0 0 1 0 0 1 1
WORD #2:    1 1 0 1 1 0 0 1 1 1 1 0 1 1 0 0
    
```

This example indicates how the NOT operator inverts the bits in a full 16-bit word. For BASIC to perform this NOT operation, the system must convert the value of word 1 to a base-10 integer. Executing the following command produces the result -9748, which is the base-10 equivalent of word 2:

```
.PRINT NOT 9747
```

F.5 TABLE OF POWERS OF TWO

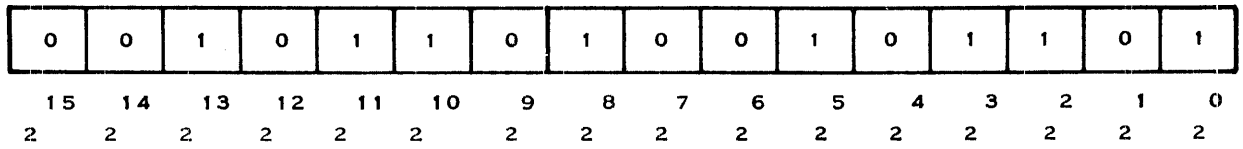
Table F-1 gives the powers-of-two values up to 2^{15} . It provides a useful aid for converting base 2 to base 10.

Table F-1. Powers of Two

N	2^N
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768

The following examples show how you can use Table F-1 to convert a base-2 integer to a base-10 integer. Notice, however, that bit 0 is a sign bit and is not equal to 2^{15} . If bit 0 is set, calculate the two's complement of the negative number. To convert a negative base-2 value to base 10, invert the full 16-bit word, add 1, calculate the base 10 equivalent as described in the preceding table, and use the negative of the result.

Logical Operators with Integer Operands



$$\begin{array}{r}
 13 \\
 1 * 2^{\quad} = 8192 \\
 1 * 2^{11} = 2048 \quad + \\
 1 * 2^{10} = 1024 \quad + \\
 1 * 2^{\quad 8} = 256 \quad + \\
 1 * 2^{\quad 5} = 32 \quad + \\
 1 * 2^{\quad 3} = 8 \quad + \\
 1 * 2^{\quad 2} = 4 \quad + \\
 1 * 2^{\quad 0} = 1 \quad + \\
 \hline
 11565
 \end{array}$$

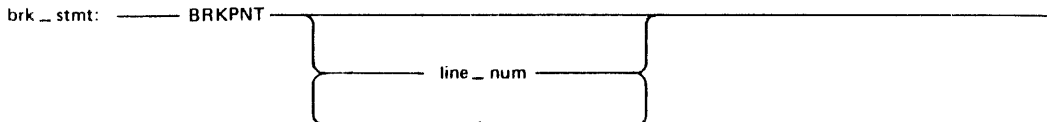
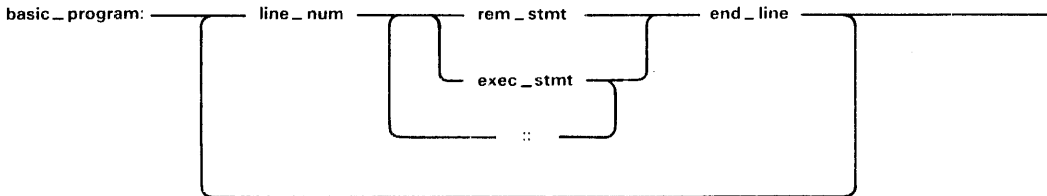
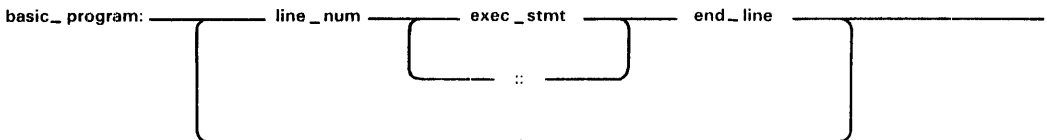
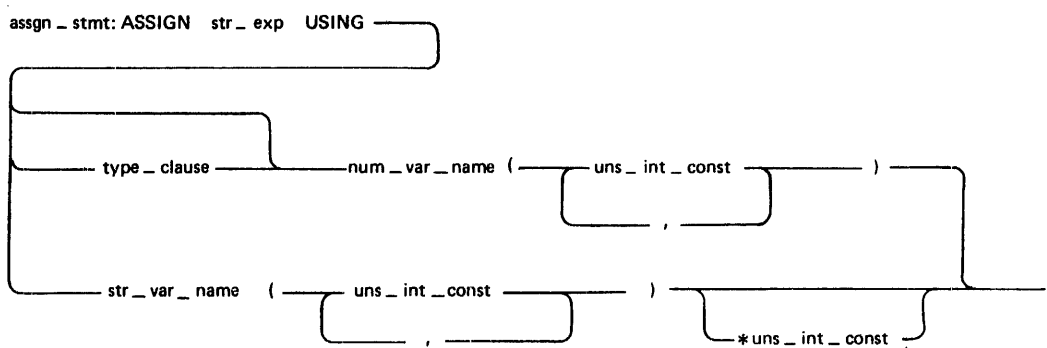
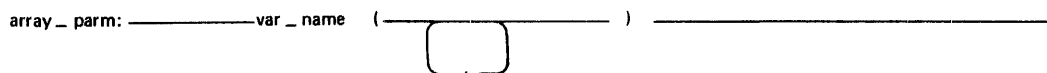
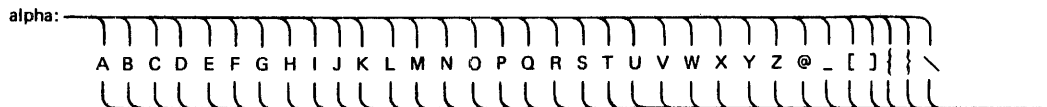
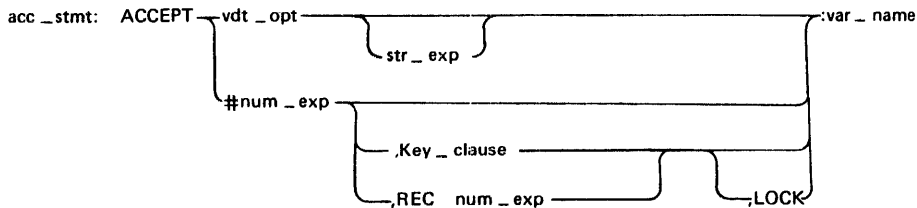
Appendix G

Syntax Diagrams

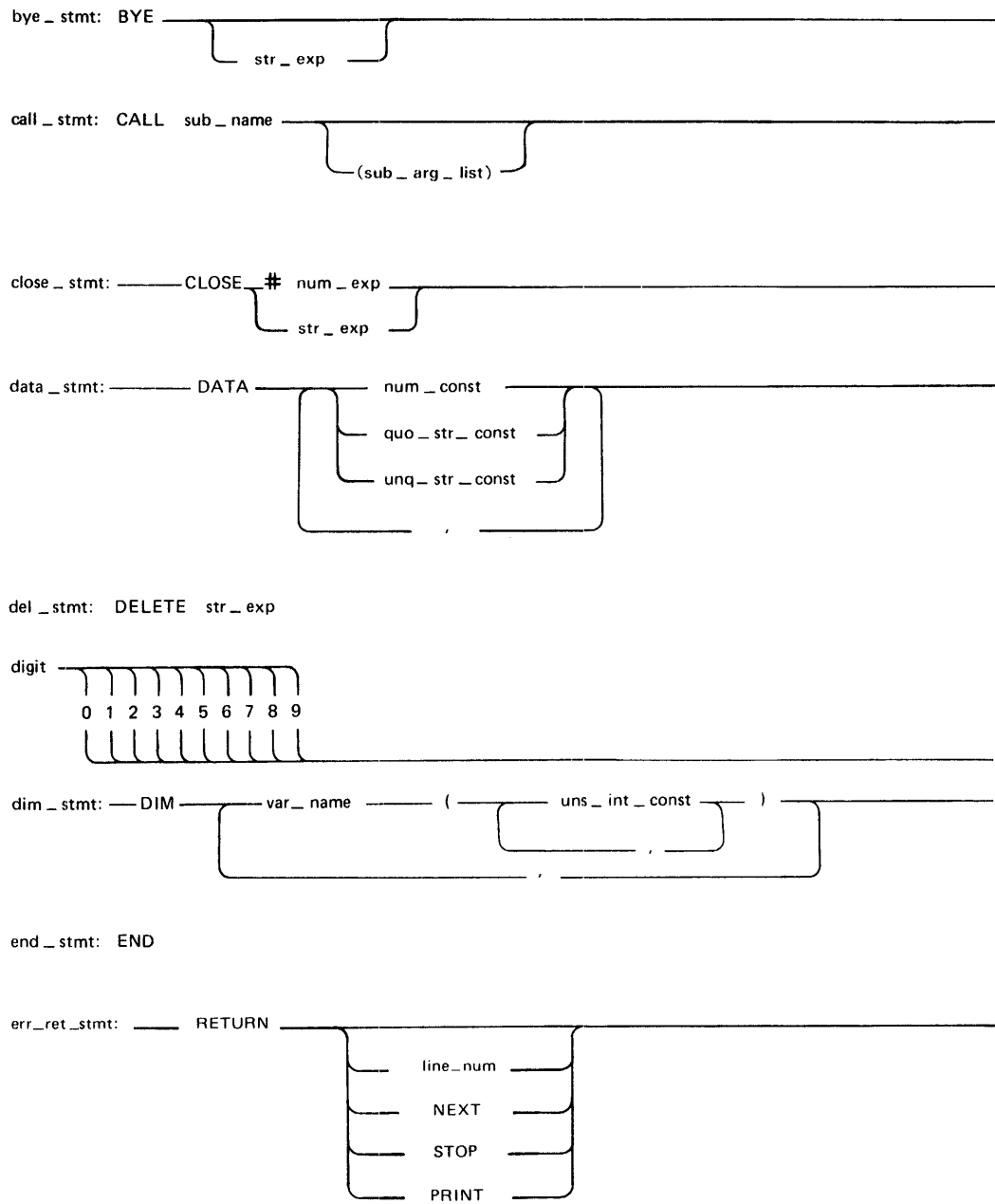
The syntax diagrams in this appendix describe the statement formats (syntax) that TI BASIC supports. The appropriate sections of this manual discuss the interstatement relationships and semantics of the language.

The syntax diagrams are presented in “railroad” normal form. The diagrams are presented in alphabetic order by diagram name. Starting on the left, trace the optional paths along the line to determine the order of the statement elements. Any capitalized words or special characters encountered must be included in the statement in the order encountered. Groups of lowercase letters, including underscores, indicate information specified in separate diagrams with that name.

Syntax Diagrams

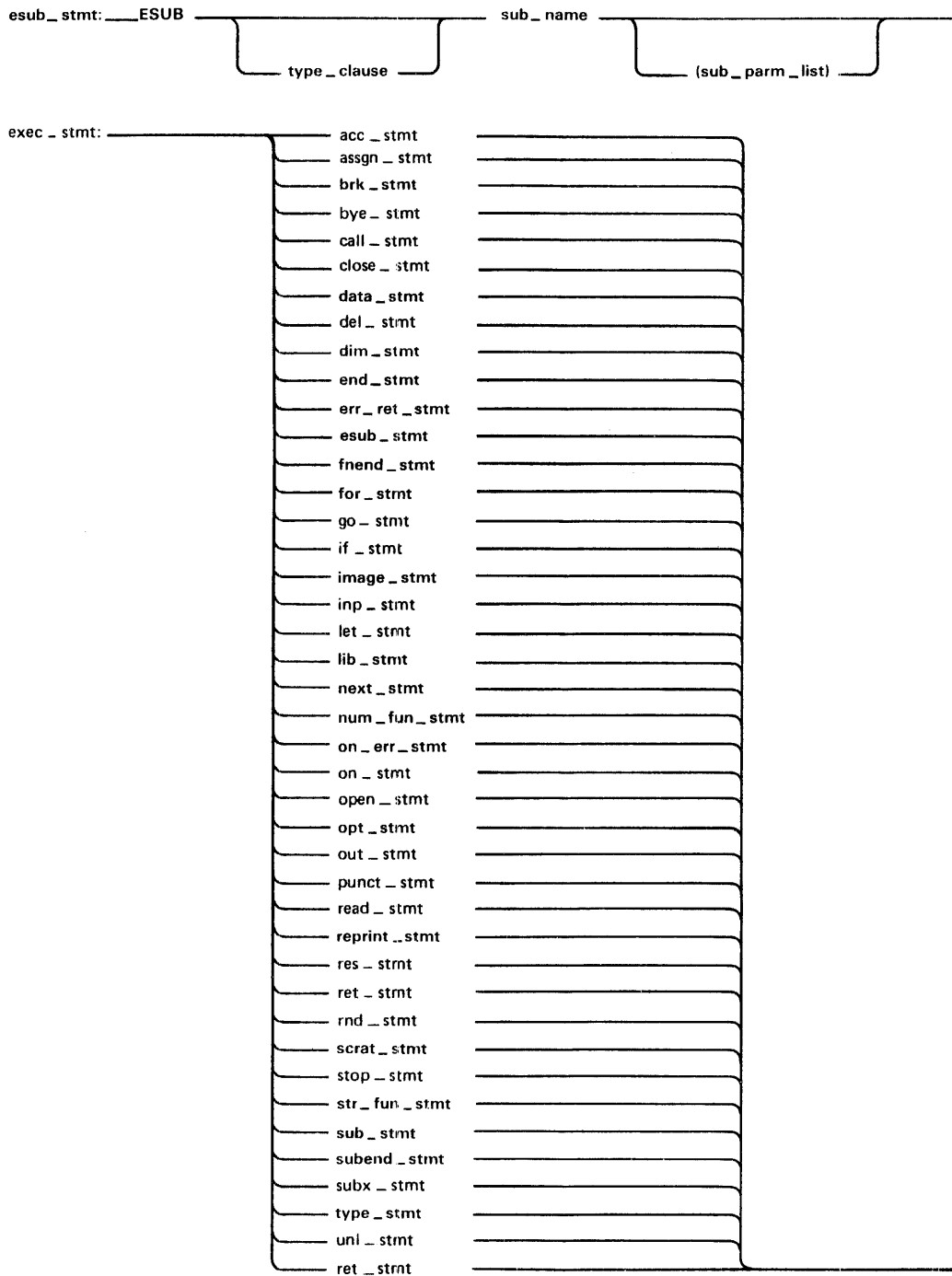


2283813 (1/10)

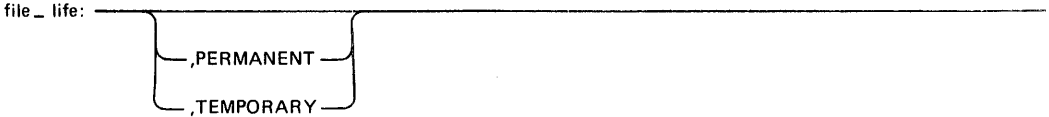
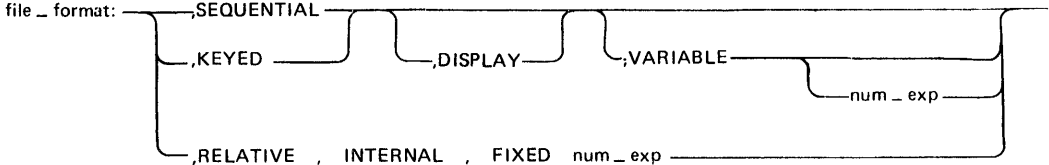
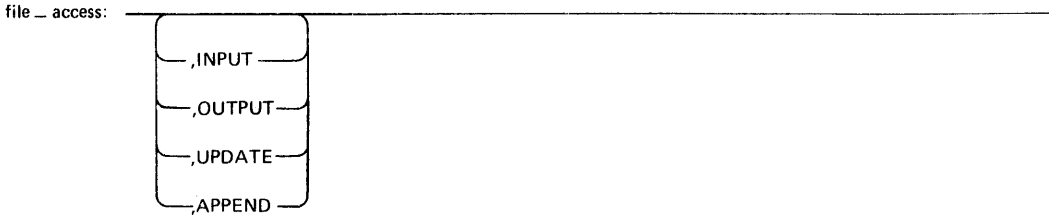


2283813 (2/10)

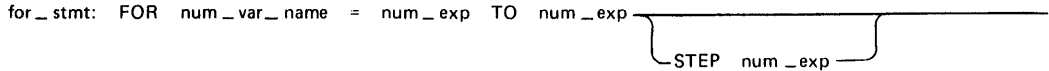
TI BASIC Syntax Diagrams (Sheet 2 of 10)



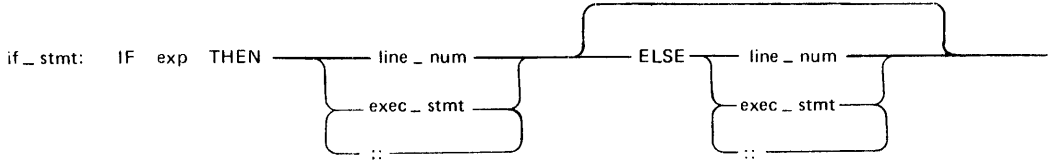
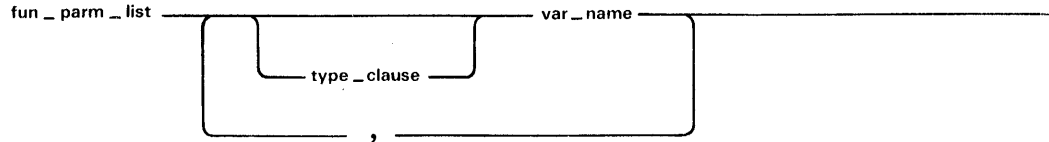
2283813 (3/10)



fnend_stmt: FNEND

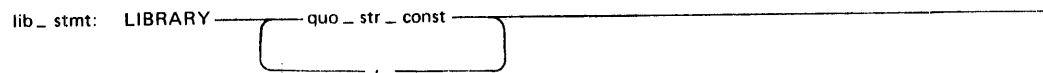
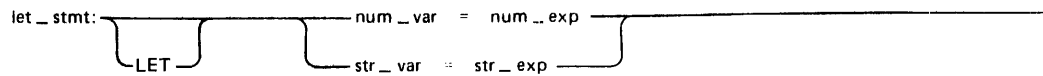
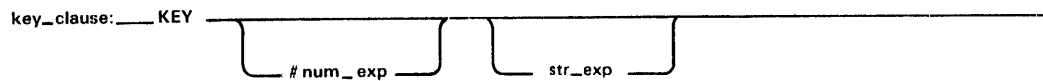
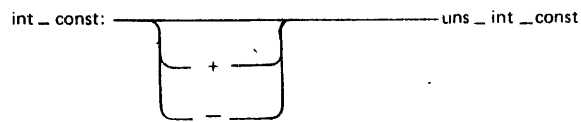
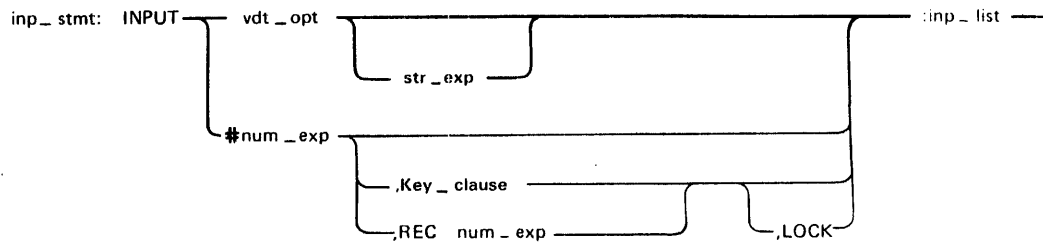
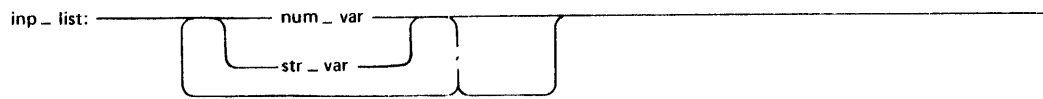


fun_name: _____ var_name _____

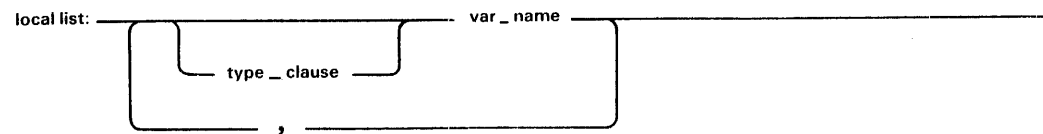


2283813 (4/10)

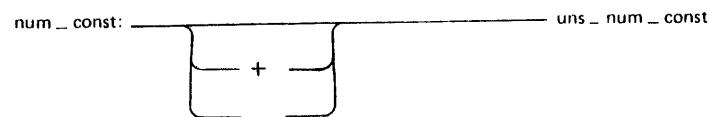
Syntax Diagrams



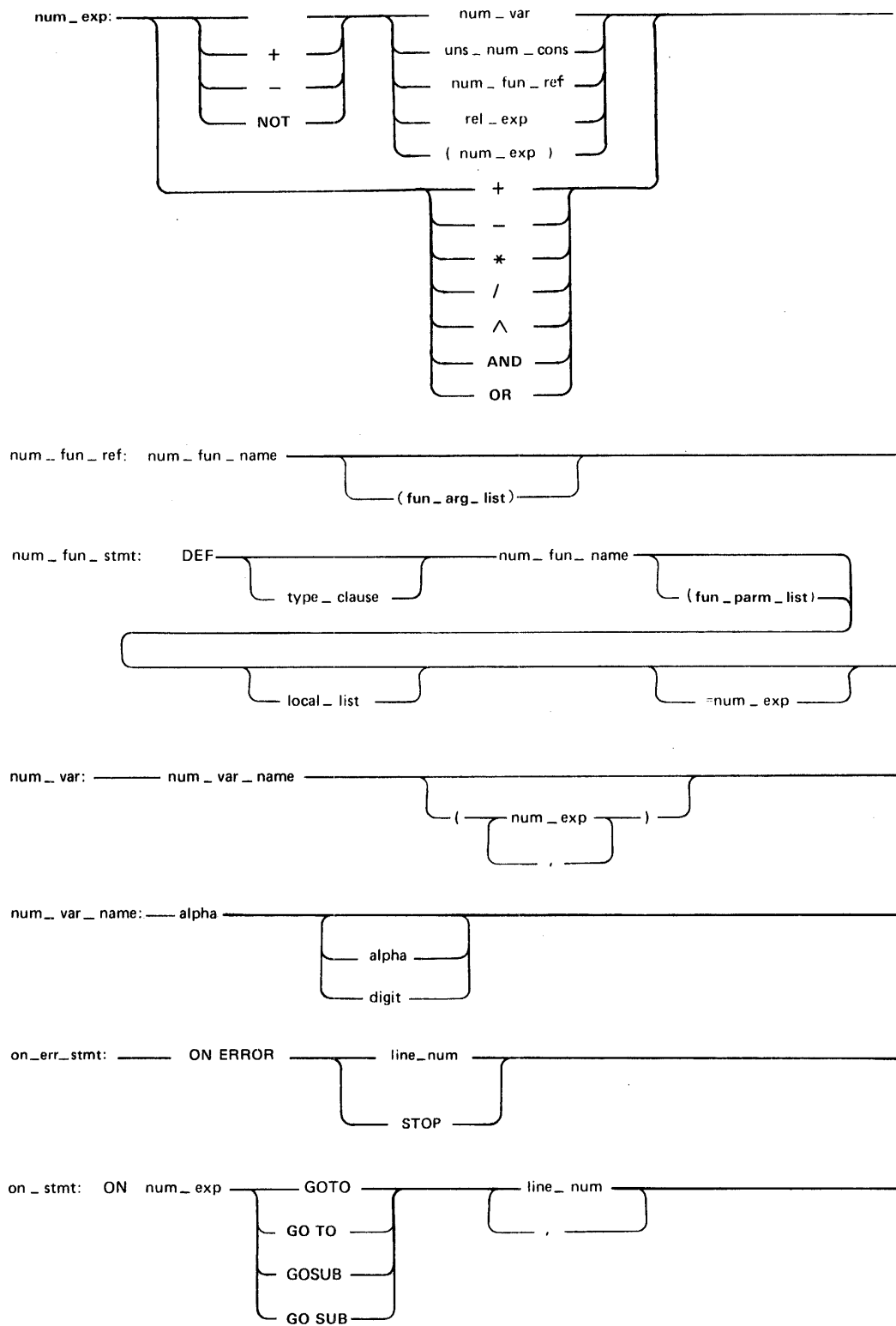
line_num: uns_int



next_stmt: NEXT num_var_name

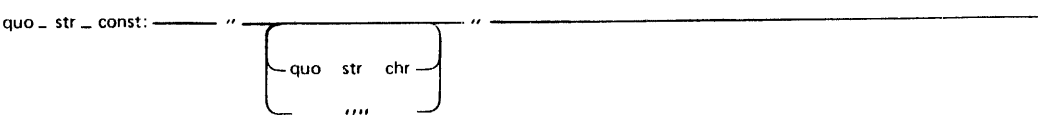
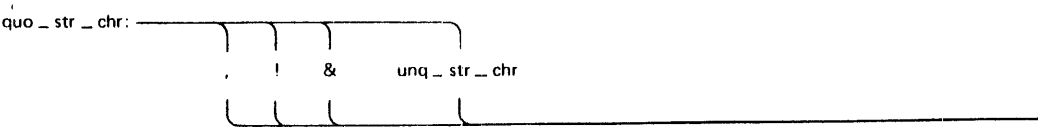
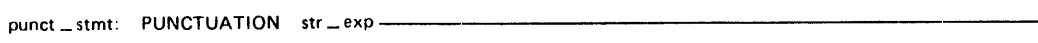
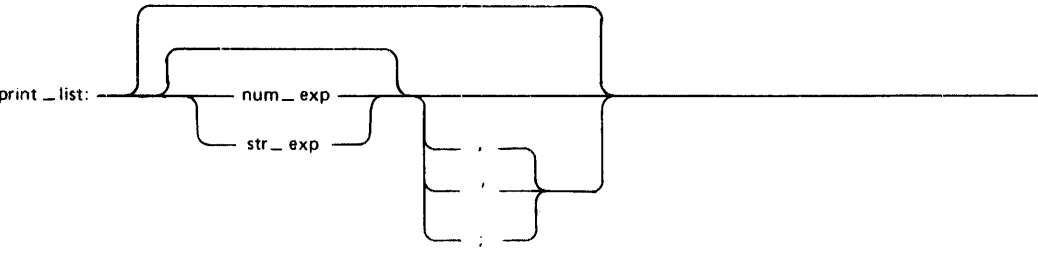
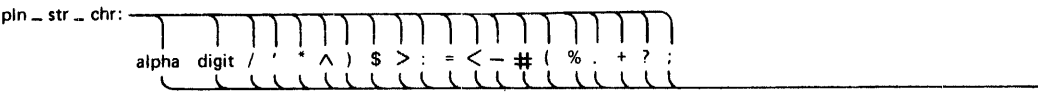
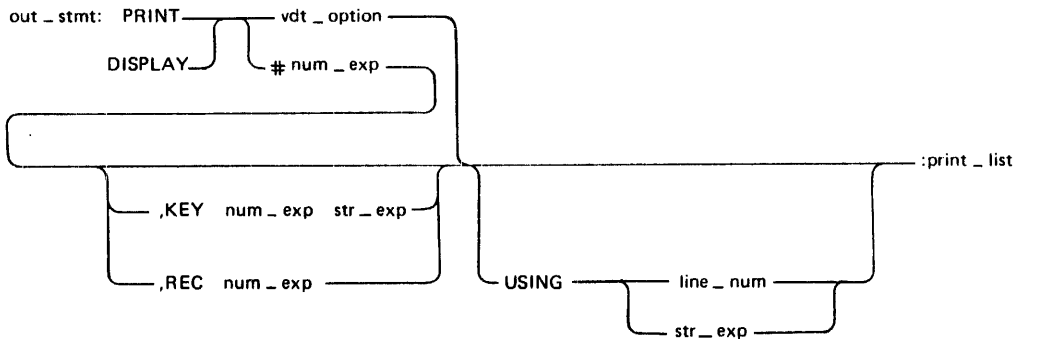
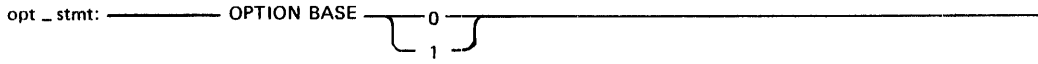
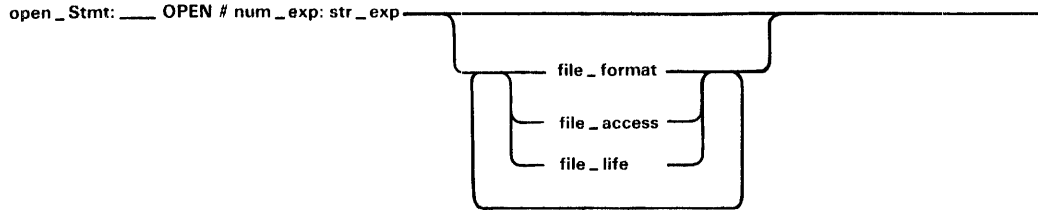


2283813 (5/10)



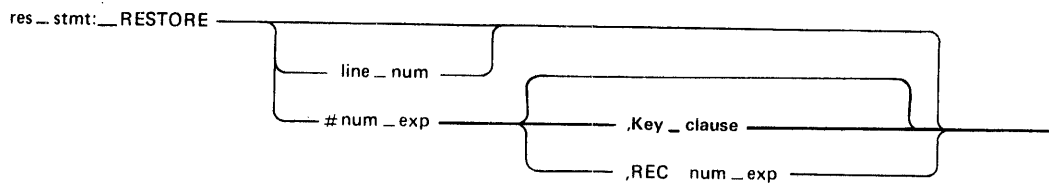
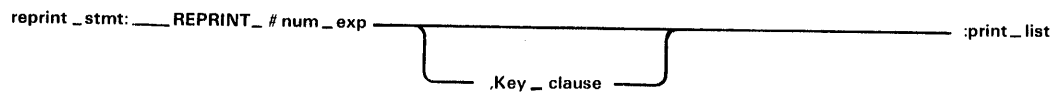
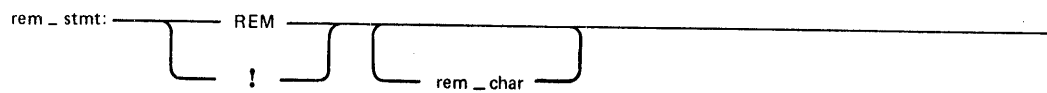
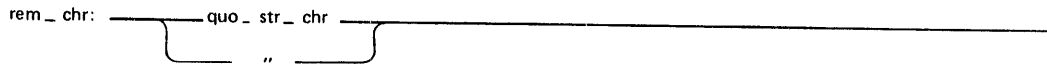
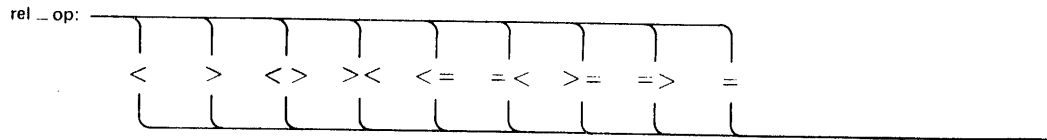
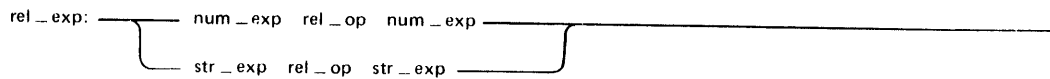
2283813 (6/10)

Syntax Diagrams

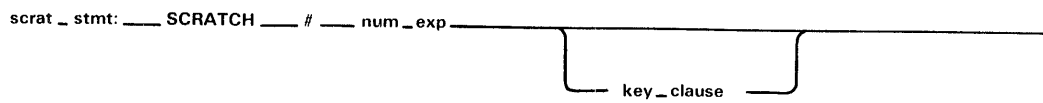


2283813 (7/10)

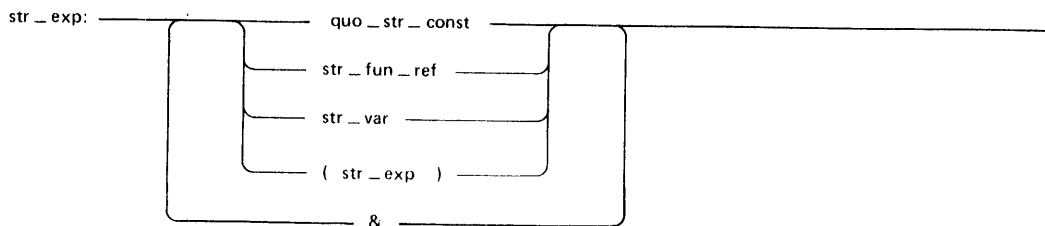
read_stmt: READ inp_list



ret_stmt: RETURN

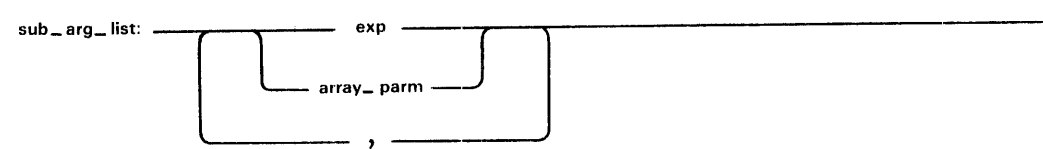
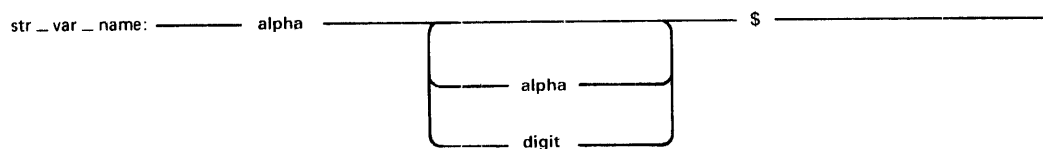
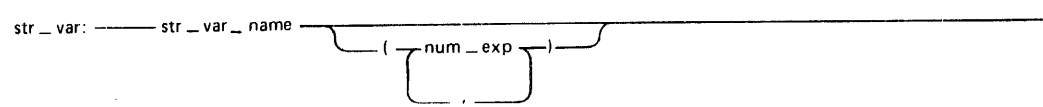
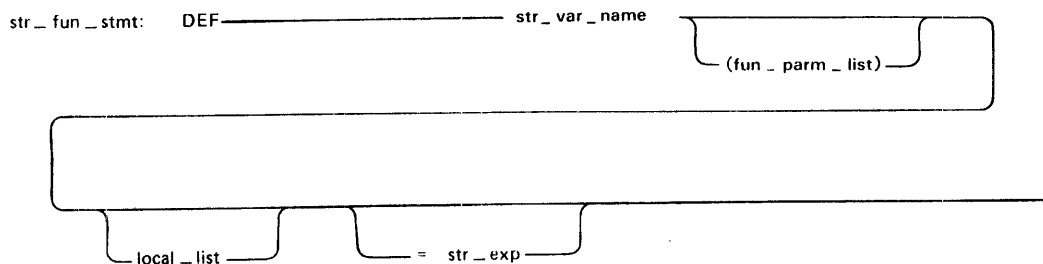
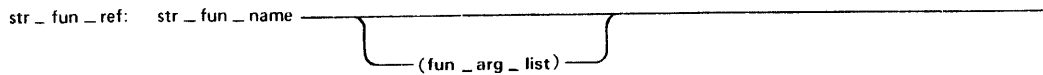


stop_stmt: STOP

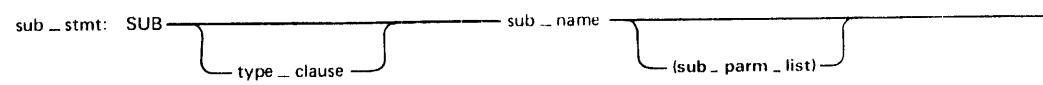
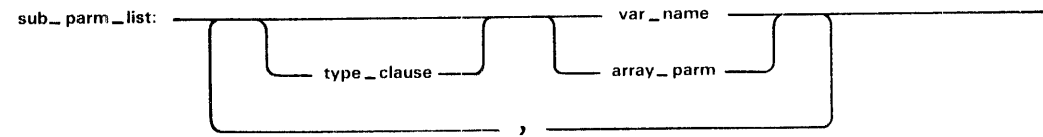


2283813 (8/10)

Syntax Diagrams



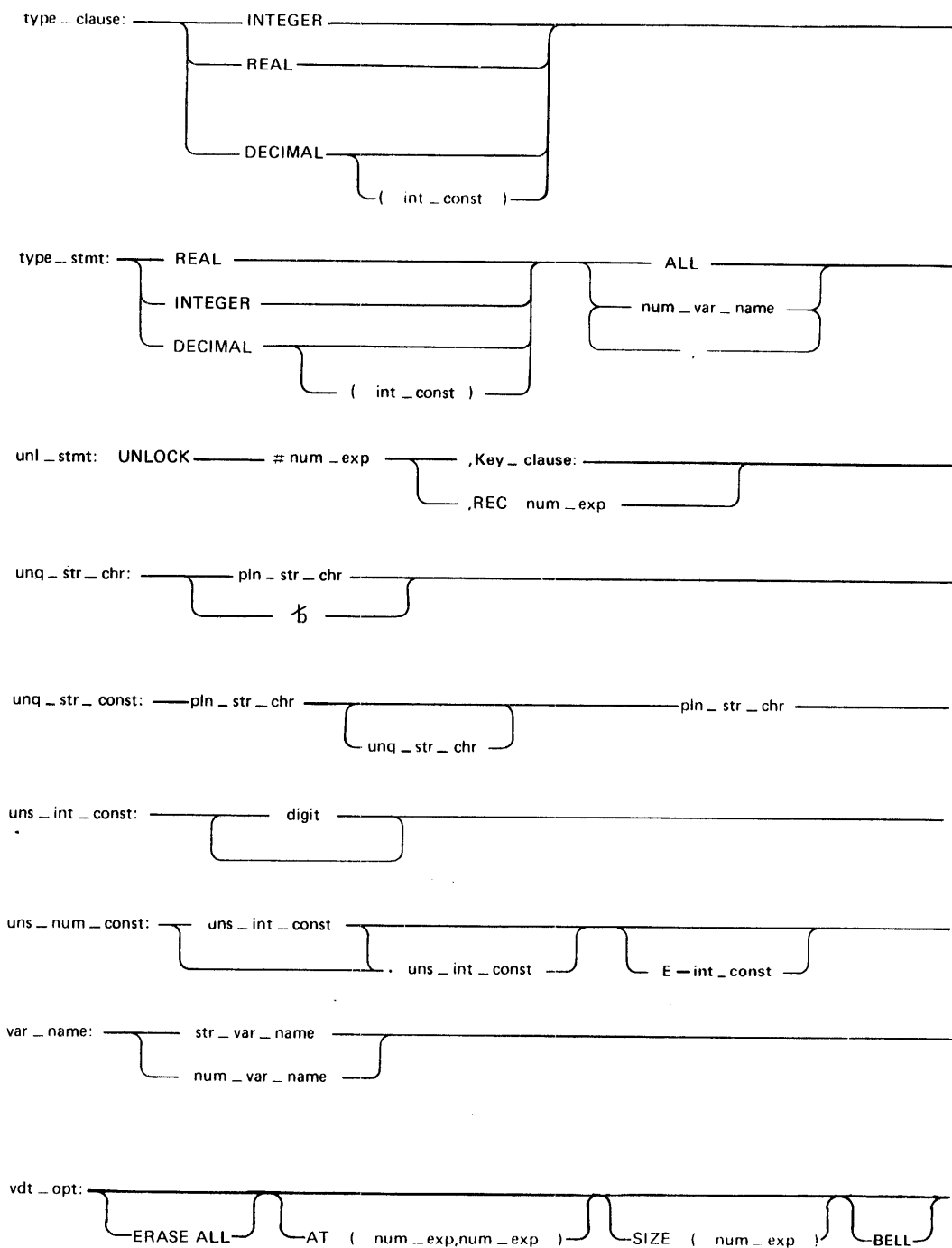
sub_name: num_var_name



subend_stmt: SUBEND

subx_stmt: SUBEXIT

2283813 (9/10)



2283813 (10/10)

Appendix H

Example Programs

H.1 INTRODUCTION

This section includes four example programs that illustrate the following:

- The use of sequential files
- The use of relative record files
- Several special features supported by TI BASIC
- The use of event keys

H.2 USE OF SEQUENTIAL FILES

The program in Figure H-1 demonstrates the use of sequential files. The program calculates squares of integer values and saves the results in a sequential file. When a maximum value is exceeded, the file is restored to reposition the internal pointer to the beginning of the file. Records from the file are then read and printed until the end-of-file is reached.

```
100 I=1 ! INITIALIZE COUNTER
110 OPEN #1: ".SQUARES",OUTPUT,INPUT
120 ! Calculate the squares of integers until limit exceeded
130 J = I*I ::IF J>1000 THEN 170
140 PRINT #1: I^J
150 I=I+1 :: J=I*I
160 GOTO 130
170 CLOSE #1 :: OPEN #1: ".SQUARES"
180 ! Print the table of squares
190 PRINT ERASE ALL "TABLE OF SQUARES LESS THAN 1000" :: PRINT
200 IF EOF(1) THEN 240
210 INPUT #1:J,K
220 PRINT TAB(5); "The SQUARE of ";J;"=";K
230 GOTO 200
240 CLOSE #1
250 END
```

Figure H-1. Example Program Using Sequential Files

H.3 USE OF RELATIVE FILES

The BASIC program in Figure H-2 illustrates relative file access in which three string values are entered. The first is hashed, or converted to a numeric value by means of a function, giving a record number within the file. The data is then stored at that record number location. To store, retrieve, or delete data from the file, enter the function codes I, F, and D, respectively.

The program handles hashing collisions (multiple values that reference the same record number) by using a linked list structure on all colliding values. To change the data filename, replace the string assignment at line 110 with the desired filename. To modify the total number of records (TABSIZ), change the assignment at line 120; use a prime number for maximum hashing efficiency.

H.4 TI BASIC SPECIAL FEATURES

Figure H-3 is an example program called HANGMAN that uses the following TI BASIC special features:

- Display with AT clause
- Display with ERASE ALL clause
- Double quotes within strings
- Double colon statement separator
- Long variable names
- String arrays
- ON-GOSUB statement
- SEG\$, POS, LEN, and UPRC\$ functions

```

100 INTEGER ALL
110 FILE$=".RELFILE"      ! RELATIVE DATA FILE
120 TABSIZ=211          ! MAXIMUM TABLE SIZE OR # OF RECORDS
130 CALL HEADER        ! CLEAR SCREEN & PRINT HEADER
140 IF FTYPE(FILE$) THEN 260 ! FILE EXISTS.
150 ! INITIAL RUN -- CREATE FILE
160 PRINT AT (12,1);"CREATING DATA FILE: "";FILE$;"" . . .";
170 OPEN #1:FILE$,OUTPUT,INPUT,RELATIVE TABSIZ,INTERNAL,FIXED 80
180 ! CREATE EMPTY RECORDS
190 VALUE=0 :: ZER$=""
200 FOR I=0 TO TABSIZ
210 PRINT #1,REC I:VALUE,ZER$,ZER$,ZER$
220 NEXT I
230 PRINT AT (12,1) "" ! ERASE MESSAGE
240 GO TO 280
250 ! OPEN FILE IF NOT FIRST RUN.
260 OPEN #1:FILE$,RELATIVE,INTERNAL,FIXED 80
270 ! DISPLAY MENU
280 PRINT AT (10,20);"NAME: "
290 PRINT AT (12,20);"ADDRESS: "
300 PRINT AT (14,20);"CITY/STATE: "
310 ! GET ENTRY
320 ACCEPT AT (10,26) SIZE(20) "" :NAME$
330 IF NAME$="" THEN CLOSE #1 :: STOP
340 ACCEPT AT (12,29) SIZE(20) "" :ADDR$
350 ACCEPT AT (14,32) SIZE(20) "" :STAT$
360 ! GET INSERT/FIND/DELETE FUNCTION
370 ACCEPT SIZE(1) "FUNCTION (I/F/D): " :FNCTN$
380 ! HASH NAME
390 ERRFLG=1 ! ASSUME AN ERROR
400 CALL HASH(ERRFLG,TABSIZ,FNCTN$,NAME$,ADDR$,STAT$)
410 IF ERRFLG THEN 480 ! ERROR IN HASH ROUTINE
420 ! DISPLAY RETRIEVED INFO
430 PRINT AT ( 9,26);NAME$
440 PRINT AT (11,29);ADDR$
450 PRINT AT (13,32);STAT$
460 ACCEPT AT (24,40) "PRESS RETURN TO CONTINUE: " :RET$
470 ! RESET SCREEN & TRY FOR ANOTHER
480 CALL HEADER :: GO TO 280
490 SUB INTEGER HEADER
500 PRINT ERASE ALL AT (2,1);"9 9 0 B A S I C H A S H R O U T I N E"
510 PRINT AT (3,1);RPT$("-",40)
520 PRINT AT (2,60);"TIME: ";TIME$ :: PRINT AT (3,60);"DATE: ";DAT$
530 SUBEND
540 SUB INTEGER HASH(ERRFLG,TABSIZ,FNCTN$,NAME$,ADDR$,STAT$)
550 ! INITIALIZE HASH PARAMETERS
560 NAMLENGTH=LEN(NAME$) :: VALUE=0 :: EOL=-1
570 ZERO=0 :: ZER$="" :: OLDREC=0
580 ! HASH NAME
590 FOR I=1 TO NAMLENGTH
600 VALUE=VALUE+ASC(SEG$(NAME$,I,1))
610 NEXT I
620 ! MOD VALUE TABLE SIZE
630 TEMP=INT(VALUE/TABSIZ)
640 RECORD=VALUE-(TEMP*TABSIZ)
650 ! GET RECORD
660 INPUT #1,REC RECORD:HSHCNT,NAM$,ADRS$,STAT$
670 IF HSHCNT=0 THEN 740 ! EMPTY RECORD
680 IF NAME$=NAM$ THEN 780 ! FOUND RECORD
690 IF HSHCNT<0 THEN 740 ! END OF LIST
700 OLDREC=RECORD ! SAVE LAST RECORD NO. READ

```

Figure H-2. Example Program Using Relative Record Files (Sheet 1 of 2)

```

710 RECORD=HSHCNT      ! GET NEXT LINK
720 GO TO 660          ! MORE COLLISIONS TO CHECK
730 ! ENTRY NOT FOUND - O.K. IF FUNCTION = INSERT
740 IF FNCTN$="I" THEN 900
750 PRINT AT (24,38) "ENTRY: ";NAME$;" :NOT FOUND: ";
760 GO TO 800
770 ! ENTRY FOUND - O.K. IF FUNCTION NEQ INSERT
780 IF FNCTN$<>"I" THEN 830
790 PRINT AT (24,38) "DUPLICATE ENTRY: ";NAME$;" ";
800 ACCEPT SIZE(1) "" :TEMP$
810 SUBEXIT           ! RETURN W/ERROR CODE SET
820 ! FUNCTION=DELETE OR FIND?
830 IF FNCTN$="D" THEN 980
840 ! FUNCTION = FIND (SET VALUES & RETURN)
850 NAME$=NAM$
860 ADDR$=ADRS$
870 STATE$=STAT$
880 GO TO 950
890 ! INSERT NEW ENTRY
900 INPUT #1,REC 0:NUMENT      ! GET # OF TABLE ENTRIES
910 IF NUMENT>=TABSIZ THEN 960 ! TABLE FULL
920 IF HSHCNT THEN CALL GETREC(RECORD,TABSIZ,NAM$,ADRS$,STAT$)
930 PRINT #1,REC RECORD:EOI,NAME$,ADRS$,STATE$
940 PRINT #1,REC 0:NUMENT+1    ! 1 MORE ENTRY
950 ERRFLG=0                  ! CLEAR ERROR FLAG
960 SUBEXIT
970 ! DELETE ENTRY- SUBTRACT 1 FROM TOTAL
980 INPUT #1,REC 0:NUMENT
990 PRINT #1,REC 0:NUMENT-1    ! 1 LESS ENTRY.
1000 ! DETERMINE POSITION OF RECORD
1010 ! W/RESPECT TO LINK LIST (PRIMARY VS. SECONDARY)
1020 IF OLDREC=0 THEN 1100    ! PRIMARY RECORD IN CHAIN
1030 ! REWRITE SECONDARY RECORD W/ADJUSTED LINK
1040 INPUT #1,REC OLDREC:VALUE,NAM$,ADRS$,STAT$
1050 PRINT #1,REC OLDREC:HSHCNT,NAM$,ADRS$,STAT$
1060 ! DELETE RECORD BY CLEARING HASH COUNT
1070 PRINT #1,REC RECORD:ZERO,ZER$,ZER$,ZER$
1080 GO TO 950
1090 ! CHECK PRIMARY RECORD FOR LINK
1100 IF HSHCNT<=0 THEN 1070   ! NO LINK- JUST ZERO HSHCNT
1110 ! IF LINK, ALWAYS SAVE PRIMARY RECORD
1120 INPUT #1,REC HSHCNT:VALUE,NAM$,ADRS$,STAT$
1130 PRINT #1,REC RECORD:VALUE,NAM$,ADRS$,STAT$
1140 RECORD=HSHCNT ! ZERO SECONDARY RECORD
1150 GO TO 1070
1160 SUBEND
1170 SUB INTEGER GETREC(RECORD,TABSIZ,NAM$,ADRS$,STAT$)
1180 ! ROUTINE TO RETURN FREE RECORD NUMBER
1190 OLDREC=RECORD
1200 FOR I=1 TO TABSIZ
1210 RECORD=RECORD+1 ! TRY NEXT LARGER RECORD NO.
1220 IF RECORD>TABSIZ THEN RECORD=1 ! MOD TABSIZ
1230 INPUT #1,REC RECORD:VALUE,TEMP$,TEMP$,TEMP$
1240 IF VALUE=0 THEN 1300 ! EMPTY RECORD FOUND
1250 NEXT I
1260 ! EMPTY RECORD SHOULD ALWAYS BE FOUND
1270 PRINT "-- FATAL ENTRY ERROR IN GETREC --"
1280 CLOSE #1 :: STOP
1290 ! SET LINK TO NEW ENTRY
1300 PRINT #1,REC OLDREC:RECORD,NAM$,ADRS$,STAT$
1310 SUBEND

```

Figure H-2. Example Program Using Relative Record Files (Sheet 2 of 2)


```

100 REM THIS PROGRAM USES THE FOLLOWING SPECIAL FEATURES:
110 REM DISPLAY WITH POSITION CLAUSE; DISPLAY WITH ERASE CLAUSE;
120 REM DOUBLE QUOTES WITHIN STRINGS; DOUBLE COLON STATEMENT SEPARATOR;
130 REM ACCEPT STATEMENT WITH POSITION-CLAUSE AND INPUT PROMPT;
140 REM LONG VARIABLE NAMES; STRING ARRAYS; ON-GOSUB STATEMENT; AND
150 REM THE FUNCTIONS SEG$, POS, LEN, AND UPRC$
160 DISPLAY ERASE ALL
170 DISPLAY AT (3,15): "INSTRUCTIONS FOR HANGMAN:"
180 DISPLAY AT (4,5): "THIS GAME OF HANGMAN ALLOWS YOU ELEVEN (11) INCORRECT"
190 DISPLAY AT (5,5): "GUESSES. TO START THE GAME, TYPE A WORD OR PHRASE WHEN"
200 DISPLAY AT (6,5): "ASKED ""WHAT PHRASE?"". THE COMPUTER WILL ERASE YOUR"
210 DISPLAY AT (7,5): "PHRASE AND PUT _ WHERE LETTERS WERE AND BLANKS BETWEEN"
220 DISPLAY AT (8,5): "WORDS OF THE PHRASE. IT WILL THEN ASK ""YOUR GUESS?"". "
230 DISPLAY AT (9,5): "IN RESPONSE, TYPE A LETTER OR YOUR GUESS OF THE WHOLE"
240 DISPLAY AT (10,5): "PHRASE. IF YOU ARE WRONG, A PIECE OF THE HANGMAN WILL"
250 DISPLAY AT (11,5): "BE PUT UP. AFTER ELEVEN WRONG GUESSES, OR WHEN YOU"
260 DISPLAY AT (12,5): "GUESS THE PHRASE, YOU WILL BE ASKED IF YOU WANT TO PLAY"
270 DISPLAY AT (13,5): "AGAIN. IF YOU DO, TYPE Y. IF YOU NEED INSTRUCTIONS"
280 ACCEPT AT (14,5) "AGAIN, TYPE I. DO YOU UNDERSTAND (Y/N)?": YES$
290 IF SEG$(YES$,1,1) = "Y" THEN 350
300 DISPLAY AT (16,5): "THERE ARE NO MORE INSTRUCTIONS, SO REREAD THESE AND"
310 DISPLAY AT (17,5): "TRY OUT THE GAME. GOOD LUCK!"
320 DISPLAY AT (19,5): "ARE YOU READY TO PLAY";
330 ACCEPT AT(19,26):YES$
340 GO TO 290
350 COMMA$ = " "
360 A$ = " "
370 DIM B$(11), POSITION(100)
380 B$(1) = "O" :: B$(2) = "I" :: B$(3) = "/" :: B$(4) = "\" :: B$(5) = " "
390 B$(6) = " " :: B$(7) = "I" :: B$(8) = "/" :: B$(9) = "\" :: B$(10) = " "
400 B$(11) = " "
410 DISPLAY ERASE ALL
420 DISPLAY AT (2,25): "H A N G M A N"
430 KOUNTOFBLANKS = 0 :: KOUNT = 0 :: B = 0 :: LETTERS_USED$ = "LETTERS USED: "
440 FOR I = 1 TO 100
450 POSITION(I) = 0
460 NEXT I
470 DISPLAY AT (5,50): "      -----"
480 DISPLAY AT (6,50): "      |          |"
490 DISPLAY AT (7,50): "      |          |"
500 DISPLAY AT (8,50): "      |          |"
510 DISPLAY AT (9,50): "      |          |"
520 DISPLAY AT (10,50): "      |          |"
530 DISPLAY AT (11,50): "      |          |"
540 DISPLAY AT (12,50): "      |////////|"
550 DISPLAY AT (13,50): "      |////////|"
560 DISPLAY AT (14,50): "      |////////|"
570 DISPLAY AT (15,50): "      |////////|"
580 ACCEPT AT (20,7) "WHAT PHRASE?": TEMP$
590 IF LEN(TEMP$) = 0 THEN 580
600 A$ = UPRC$(TEMP$)
610 X = LEN(A$)
620 DISPLAY AT (20,6+X): "          "
630 DISPLAY AT (20,7): "          "
640 FOR I = 1 TO X :: DISPLAY AT (20,9+I) SIZE (1): "_";: NEXT I
650 C$ = " "
660 Y = 1
670 Z = POS(A$,C$,Y)
680 IF Z = 0 THEN 740
690 DISPLAY AT (20,9+Z) SIZE (1): " ";
700 Y = Z + 1
710 KOUNTOFBLANKS = KOUNTOFBLANKS + 1

```

Figure H-3. Example Program, HANGMAN (Sheet 1 of 3)

```

720 GO TO 670
730 IF KOUNTOFBLANKS + KOUNT = X THEN 980
740 ACCEPT AT (22,7) "YOUR GUESS?": T$
750 IF LEN(T$) = 0 THEN 740
760 D$ = UPRC$(T$)
770 DISPLAY AT (23,10): "
780 IF LEN(D$) = 1 THEN 1010
790 IF LEN(D$) = X THEN 950
800 R = 1
810 SEGPOS = POS(A$,D$,R)
820 IF SEGPOS = 0 THEN IF R = 1 THEN 960 ELSE 890 ELSE 830
830 DISPLAY AT(20,9+SEGPOS)SIZE (LEN(D$)): SEG$(TEMP$,SEGPOS,LEN(D$));
840 R = SEGPOS + LEN(D$)
850 FOR I = SEGPOS TO LEN(D$)+SEGPOS-1
860 POSITION(I) = 1
870 NEXT I
880 GO TO 810
890 KOUNT = 0
900 FOR I = 1 TO X
910 KOUNT = KOUNT + POSITION(I)
920 NEXT I
930 IF KOUNTOFBLANKS + KOUNT = X THEN 980
940 GO TO 1220
950 IF D$ = A$ THEN 980
960 B = B + 1 :: DISPLAY AT (23,10): "WRONG PHRASE" :: GOSUB 1460
970 GO TO 1230
980 DISPLAY AT (23,10): "CONGRATULATIONS, YOU WON WITH";B;"INCORRECT GUESSES"
990 GO TO 1290
1000 DISPLAY AT(22,5): "
1010 CHECK = POS(LETTERS_USED$,D$,13)
1020 IF CHECK <> 0 THEN 1160
1030 C = 1
1040 D = POS(A$,D$,C)
1050 IF D = 0 THEN IF C = 1 THEN 1190 ELSE 1210 ELSE 1060
1060 X$ = SEG$(TEMP$,D,1)
1070 DISPLAY AT (20,9+D)SIZE (1): X$;
1080 POSITION(D) = 1
1090 C = D + 1
1100 KOUNT = 0
1110 FOR I = 1 TO X
1120 KOUNT = KOUNT + POSITION(I)
1130 NEXT I
1140 IF KOUNTOFBLANKS + KOUNT = X THEN 980
1150 GO TO 1040
1160 DISPLAY AT (23,10): " YOU´VE TRIED THAT ALREADY!"
1170 B = B + 1 :: GOSUB 1460
1180 GO TO 1230
1190 DISPLAY AT (23,10): "THAT LETTER ISN´T IN THE PHRASE"
1200 B = B + 1 :: GOSUB 1460
1210 LETTERS_USED$ = LETTERS_USED$ & COMMA$ & D$
1220 DISPLAY AT (24,1): LETTERS_USED$
1230 IF B = 11 THEN 1260
1240 DISPLAY AT (22,1): "
1250 GO TO 740
1260 DISPLAY AT (23,25): "
1270 DISPLAY AT (23,10): "YOU LOSE!"
1280 DISPLAY AT (22,5): "
1290 DISPLAY AT (20,10): TEMP$
1300 DISPLAY AT (24,1): "DO YOU WANT TO PLAY AGAIN";
1310 INPUT ANSWER$
1320 IF SEG$(ANSWER$,1,1) = "Y" THEN 410
1330 IF SEG$(ANSWER$,1,1) = "I" THEN 160

```

Figure H-3. Example Program, HANGMAN (Sheet 2 of 3)

```

1340 RUN "DS02.DEMO"
1350 DISPLAY AT (7,57) SIZE (1):B$(1);: RETURN
1360 DISPLAY AT (8,57) SIZE (1): B$(2);: RETURN
1370 DISPLAY AT (8,56) SIZE (1): B$(3);: RETURN
1380 DISPLAY AT (8,58) SIZE (1): B$(4);: RETURN
1390 DISPLAY AT (8,55) SIZE (1): B$(5);: RETURN
1400 DISPLAY AT (8,59) SIZE (1): B$(6);: RETURN
1410 DISPLAY AT (9,57) SIZE (1): B$(7);: RETURN
1420 DISPLAY AT (10,56) SIZE (1): B$(8);: RETURN
1430 DISPLAY AT (10,58) SIZE (1): B$(9);: RETURN
1440 DISPLAY AT (10,55) SIZE (1): B$(10);: RETURN
1450 DISPLAY AT (10,59) SIZE (1): B$(11);: RETURN
1460 ON B GOSUB 1350,1360,1370,1380,1390,1400,1410,1420,1430,1440,1450
1470 RETURN
1480 END

```

Figure H-3. Example Program, HANGMAN (Sheet 3 of 3)

H.5 EVENT KEY IMPLEMENTATION

The program in Figure H-4 illustrates the use of an event key. Event keys allow you to interrupt execution of a program by entering a function key. In this program, entering a Command or Attention key generates error number 1. The ON ERROR statement traps this error and causes the program to query you for further instructions. These instructions are accepted through the ACCEPT statement; however, they could be input by either an INKEY\$ function or an INPUT statement. After you have executed the event function, you can resume execution of the main program by executing the RETURN NEXT command.

H.6 KEY-INDEXED FILE EXAMPLE

The program in Figure H-5 shows an example of the use of key indexed files (KIFs) with BASIC. Prior to the execution of this program, the file BASIC.KEY was generated using the Create Key Indexed File (CFKEY) command. The logical record length was set to 64, the physical record length was set to 300, and the initial and secondary allocations were defaulted. The maximum size was set to 100. The primary key was started in position 1, given a length of 20, and specified nonmodifiable with no duplicates. The second key was started in position 22 with a length of 20 and modification and duplicates permitted. Note that position 21 was not allocated to provide space for the comma that BASIC generates when multiple data items are printed using the apostrophe separator. Similarly, the third key begins in position 43 and in all other aspects is identical to the second key.

This program allows you to add records to a KIF, locate records within the file on any of the three keys, and delete records from the file.

```

100 OPEN #1:"BASIC.KEY",KEYED
110 PRINT ERASE ALL AT (2,18);" B A S I C   K E Y   R O U T I N E"
120 PRINT AT (10,30);"NAME: " :: PRINT AT (12,30);"CITY: "
130 PRINT AT (14,30);"STATE: "
140 ACCEPT AT (10,37) SIZE(20) "" :NAME$
150 IF NAME$ = "STOP" THEN CLOSE #1 :: STOP
160 CALL FILL(NAME$)
170 ACCEPT AT (12,37) SIZE(20) "" :CITY$
180 CALL FILL(CITY$)
190 ACCEPT AT (14,37) SIZE(20) "" :STATE$
200 CALL FILL(STATE$)
210 ACCEPT AT(24,1) SIZE(1) "KEY NUMBER: ":K :: IF K<1 OR K>3 THEN 210
220 ACCEPT AT (16,20) SIZE(1) "I-INSERT,D-DELETE,F-FIND: ":Y$
230 IF Y$ = "I" THEN 390 ELSE IF Y$ = "D" THEN 430
240 IF Y$ <> "F" THEN 110
250 ON K GO TO 300,280,260
260 INPUT #1,KEY #3 STATE$:A$,B$,C$
270 GO TO 330
280 INPUT #1,KEY #2 CITY$:A$,B$,C$
290 GO TO 330
300 INPUT #1, KEY#1 NAME$:A$,B$,C$
310 GO TO 330
320 INPUT #1:A$,B$,C$
330 IF EOF(1) THEN 370
340 PRINT AT (10,37);A$ :: PRINT AT (12,37);B$ :: PRINT AT (14,37);C$
350 ACCEPT AT (16,20) "RETURN - CONTINUE/N - NEXT: ":Y$
360 IF Y$ <> "N" THEN 110 ELSE 320
370 ACCEPT AT (16,20) "NO ENTRY! --- ENTER? (Y/N): ":Y$
380 IF Y$ <> "Y" THEN 110
390 PRINT #1,KEY #1 NAME$:NAME$`CITY$`STATE$
400 IF DUP(1) = 0 THEN 110
410 ACCEPT AT (16,14) "DUPLICATE NAME! -- DELETE PREVIOUS ENTRY? (Y/N): ":Y$
420 IF Y$ <> "Y" THEN 110
430 SCRATCH #1,KEY NAME$ :: GO TO 110
440 SUB FILL(TEMP$)
450 ! PAD KEY WITH BLANKS TO THE RIGHT
460 IF LEN(TEMP$) < 20 THEN TEMP$ = TEMP$ & RPT$(" ",20-LEN(TEMP$))
470 SUBEND
480 END

```

Figure H-4. Sample KIF Program

Appendix I I

BASIC Systems Function Keys

Table I-1.

BASIC Function	911, S200 or S300 Key	940 Key	ASR/KSR
Space Forward	Space Bar	Space Bar	Space Bar
Move Cursor Right	Right Arrow Right CHAR	Right Arrow	—
Move Cursor Left	Left Arrow Left CHAR	Left Arrow BACK SPACE	BACKSPACE
Back Tab	Left FIELD	SKIP Left	—
Return	RETURN	RETURN	RETURN
Return with EOF	ENTER	SEND	CTRL S
Display Current or Preceding Line	Up Arrow	Up Arrow	—
Display Current or Succeeding Line	Down Arrow	Down Arrow	—
Insert Character	INS CHAR	INS CHAR	—
Delete Character	DEL CHAR	DEL CHAR	—
Erase Input	ERASE INPUT	ERASE INPUT	RUB OUT DEL (KSR)
Erase Field	SKIP	SKIP Right	—
Tab	TAB	TAB Right	—
Repeat	REPEAT Typamatic	Typamatic	—
Replay	Blank Gray F2	INS LINE F2	—
Calculate	F1	F1	CTRL A

Table I-1. (Continued)

BASIC Function	911, S200 or S300 Key	940 Key	ASR/KSR
Break Execution	Blank Orange, CTRL X	Blank Orange, CTRL X	ESC/ CTRL X
Resume Execution	F7	F7	CTRL V
Step	F8	F8	CTRL W
Reset Cursor	HOME	HOME	—
Suspend Output	Blank Orange	PREV FORM	ESC

The following keys have no special functions in BASIC. Some keys have functions in SCI; others can be assigned functions as desired. The function codes returned by the operating system are as follows:

Command	CMD	NEXT FORM	CMD
Erase	ERASE FIELD	ERASE EOF	—
Print	PRINT	PRINT	
	F3	F3	F3
	F4	F4	F4
	F5	F5	F5
	F6	F6	F6

Alphabetical Index

Introduction

HOW TO USE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables.

INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections — Reference to Sections of the manual appear as “Sections x” with the symbol x representing any numeric quantity.
- Appendixes — Reference to Appendixes of the manual appear as “Appendix y” with the symbol y representing any capital letter.
- Paragraphs — Reference to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph may be found.
- Tables — References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number.

Tx-yy

- Figures — References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number.

Fx-yy

- Other entries in the Index — References to other entries in the index preceded by the word “See” followed by the referenced entry.

- Absolute Value (ABS) Function 8.2.1
- ACCEPT Statement 6.6, 6.6.4
- Access Mode:
 - APPEND 6.3.5.4
 - File 6.3.5
 - INPUT 6.3.5.2
 - OUTPUT 6.3.5.1
 - UPDATE 6.3.5.3
- Accessing:
 - Array 11.8
 - Numeric Array 11.8.1
 - String Array 11.8.2
- Alphanumeric Field 6.5.5.5
- ANSI Enhancements 1.1
- Apostrophe Data Separator 6.5.2.3
- APPEND Access Mode 6.3.5.4
- Arctangent (ATN) Function 8.2.2
- Arithmetic:
 - Expression 5.11.1
 - Operator 5.9.1, T5-1
- Array 5.6
 - Accessing 11.8
 - Numeric 11.8.1
 - String 11.8.2
 - Virtual 5.7
- ASCII Character Set Appendix B
- ASCII Character/Graphic Character
 - Codes TB-1
- ASCII to Decimal (ASC) Function 8.3.1
- Assembly Language:
 - Subroutine Section 11
 - Creation 11.2
 - Linking 11.3
- ASSIGN Statement 5.7.1
- AT Option 6.5.3.2
- Back Tab Function Key 4.8.3
- BASIC:
 - Command 2.1, 2.2.1
 - Commands Section 3
 - Elements 1.3
 - Function Keys T1-2
 - Intrinsic Function T8-1
 - Subroutine Library Section 12
 - Subroutine Arguments 1.2.2.1
 - Subroutine Error Codes 1.2.2.2
 - Supported Devices T1-1
- BELL Option 6.5.3.4, E-6
- Bit Positions of Function Key T4-1
- BLDBUF 1.4.6.1
- Break:
 - Execution Function Key 4.8.15
 - Function Key 10.2.2
- Break (BREAK) Function 8.3.2
- BRKPNT Command 10.2.1
- BYE Command 2.1, 2.2.5, 3.14
- Calculate Function Key 4.8.14
- CALL:
 - Statement 9.3.1, 11.5
 - Variables in 11.5
- Character:
 - Function Key:
 - Delete 4.8.8
 - Insert 4.8.7
 - Set, ASCII Appendix B
 - String, Data 5.2
 - Character (CHR\$) Function 8.3.10
 - Characteristics, File E.3
 - Characters in Buffer (INKEY)
 - Function 8.5.4
 - Clause, KEY 6.6.3.3
 - CLOSE Statement 5.7.2, 6.4
 - Code Example, Source F11-6
 - Codes:
 - ASCII Character/Graphic
 - Character TB-1
 - Error Appendix D
 - Comma Data Separator 6.5.2.1
 - Command:
 - BASIC 2.1, 2.2.1
 - BRKPNT 10.2.1
 - BYE 2.1, 2.2.5, 3.14
 - DELETE 3.8, 4.6
 - ESUB 3.8.2
 - File 3.8.1
 - Lines 3.8.3
 - EDIT 3.6
 - ESUB 3.6.2
 - MAIN 3.6.1
 - LIST 3.7, 4.4
 - MERGE 3.10, 4.7
 - NEW 2.2.2, 3.2, 6.4
 - NUM 2.2.2, 3.3
 - OLD 3.5, 4.4, 4.5, 6.4
 - RENAME 3.9
 - RESEQUENCE 3.4, 4.5
 - RUN 2.2.4, 3.13, 6.4
 - SAVE 2.2.2, 3.12
 - TRACE 10.5
 - UNBRKPNT 10.2.1
 - UNTRACE 10.5
 - UPDATE 3.11
 - Commands, BASIC Section 3
 - Computed GOSUB Statement 7.7
 - Configuration System 1.2
 - Constant 5.3
 - Numeric 5.3.1
 - String 5.3.2
 - Control Statement Section 7
 - Cosine (COS) Function 8.2.3
 - Creation, Assembly Language
 - Subroutine 11.2
 - Cross-Reference Example, Object
 - Listing F11-8
 - Current Line Function Key,
 - Display 4.8.5, 4.8.6
 - Cursor Left Function Key 4.8.2
 - Cursor Right Function Key 4.8.2
- Data Section 5
 - Character String 5.2

- Data Entry, KIF 6.6.3.3
- Data Format:
 - Decimal 11.7.3
 - Integer 11.7.1, F11-3
 - Real 11.7.2, F11-4
 - String 11.7.4, F11-5
- Data Numeric 5.2
- Data Separator 6.5.2
 - Apostrophe 6.5.2.3
 - Comma 6.5.2.1
 - Semicolon 6.5.2.2
- DATA Statement 6.7.1
- Data Values, Parameter 11.7
- Date (DAT\$) Function 8.4.1
- Debugging Section 10
- Decimal:
 - Data Format 11.7.3
 - Field 6.5.5.3
- DECIMAL Statement 5.4.3.2
- DEF Statement Variables in 9.5
- Define (DEF) Statement 9.2, 9.2.1
- Delete Character Function Key 4.8.8
- DELETE:
 - Command 3.8, 4.6
 - ESUB Command 3.8.2
 - File Command 3.8.1
 - Lines Command 3.8.3
- Device Output 6.5.1
- Devices, BASIC Supported T1-1
- Diagrams, Syntax Appendix G
- DIM Statement 5.6.1
- Display Current Line Function Key 4.8.5, 4.8.6
- DISPLAY Format 6.3.2.1
- Display Preceding Line Function Key 4.8.5
- DISPLAY Statement 6.5
- Display Succeeding Line Function Key 4.8.6
- DNOS:
 - Operating System E.1
 - Pathnames E.3.3.1
 - Spooler Devices E.7
- DXM:
 - File Restrictions E.3.2
 - Operating System E.1
 - Pathnames E.3.3.2
 - Program Size Restriction E.4
- DX10:
 - Operating System E.1
 - Pathnames E.3.3.1
- EDIT:
 - Command 3.6
 - ESUB Command 3.6.2
- Edit Function 4.8
- EDIT MAIN Command 3.6.1
- Edit Program 4.4
- Editing Capabilities Section 4
- End of File (EOF) Function 8.5.6
- END Statement 7.9
- Enhancements, ANSI 1.1
- ERASE ALL Option 6.5.3.1
- Erase:
 - Field Function Key 4.8.10
 - Input Function Key 4.8.9
- ERR Function 7.8, 8.5.9
- Error:
 - Codes Appendix D
 - Messages Appendix D, TD-1
- Errors, Input 6.6.2.6
- ESUB:
 - Command:
 - DELETE 3.8.2
 - EDIT 3.6.2
 - Statement 9.3.2.2
- Evaluation, Expression 5.11
- Example Program Appendix H
- Exclamation Point — See Remark
- Statement 4.3.2
- Execution:
 - Function Key:
 - Break 4.8.15
 - Resume 4.8.16, 10.3
 - Step by Step 10.4
- Exponential (EXP) Function 8.2.4
- Exponential Field 6.5.5.4
- Expression Section 5
 - Arithmetic 5.11.1
 - Evaluation 5.11
 - Logical 5.11.2
 - Relational 5.11.4
 - String 5.11.3
- Expressions 5.8
- External Subprogram 9.3
- Field:
 - Alphanumeric 6.5.5.5
 - Decimal 6.5.5.3
 - Exponential 6.5.5.4
 - Function Key, Erase 4.8.10
 - Integer 6.5.5.2
 - Literal 6.5.5.6
- File:
 - Access Mode 6.3.5
 - Characteristics E.3
 - Command, DELETE 3.8.1
 - Format 6.3.2
- INPUT:
 - Statement with Relative Record 6.6.3.2
 - Statement with Sequential 6.6.3.1
- Life 6.3.4
- OPEN:
 - Statement with, Relative
 - Record 6.3.1.2
 - Statement with, Sequential 6.3.1.1
- Output 6.5.6
 - Relative Record 6.5.6.2
 - Sequential 6.5.6.1
- Record Length 6.3.3
- Relative Record 6.2.2
- Restrictions, DXM E.3.2
- Sequential 6.2.1
- Shared 6.9

File (See KIF), Key Indexed	6.3.1.3
Find Available Space (FREESPACE) Function	8.5.3
Fixed Record Length	6.3.3.2
FOR Statement	7.5
Format Control Characters	T6-1
Format:	
DISPLAY	6.3.2.1
File	6.3.2
INTERNAL	6.3.2.2
Function:	
Absolute Value (ABS)	8.2.1
Arctangent (ATN)	8.2.2
ASCII to Decimal (ASC)	8.3.1
BASIC Intrinsic	T8-1
Break (BREAK)	8.3.2
Character (CHR\$)	8.3.10
Characters in Buffer (INKEY)	8.5.4
Cosine (COS)	8.2.3
Date (DAT\$)	8.4.1
End (FNEND) Statement	9.2, 9.2.2
End of File (EOF)	8.5.6
ERR	7.8, 8.5.9
Exponential (EXP)	8.2.4
Find Available Space (FREESPACE)	8.5.3
Integer (INT)	8.2.5
Intrinsic	Section 8
Key:	
Back Tab	4.8.3
Bit Positions of	T4-1
Break	10.2.2
Execution	4.8.15
Calculate	4.8.14
Cursor Left	4.8.2
Cursor Right	4.8.2
Delete Character	4.8.8
Display:	
Current Line	4.8.5, 4.8.6
Preceding Line	4.8.5
Succeeding Line	4.8.6
Erase:	
Field	4.8.10
Input	4.8.9
Insert Character	4.8.7
Repeat	4.8.12
Replay	4.8.13
Reset Cursor	4.8.18
Resume Execution	4.8.16, 10.3
Return	4.8.4
Return with EOF	4.8.20
Space Forward	4.8.1
Step	4.8.17, 10.4
Suspend Execution	4.8.19
Tab	4.8.11
Values Associated with	T4-1
Length (LEN)	8.3.3
Match String (SPAN)	8.3.7
Mathematical	8.2
Natural Logarithm (LOG)	8.2.6
Numeric (NUM)	8.3.4
Position (POS)	8.3.5
Random Number (RND)	8.5.1
Recursive	9.2.3
Repeat (RPT\$)	8.3.6
Return Character (INKEY\$)	8.5.5
Segment (SEG\$)	8.3.11
Sign (SGN)	8.2.7
Sine (SIN)	8.2.8
Square Root (SQR)	8.2.9
String	8.3
String (STR\$)	8.3.12
Tab (TAB)	8.5.8
Tangent (TAN)	8.2.10
Test for Duplicate Keys (DUP)	8.5.10
Time (TIME\$)	8.4.2
Uppercase (UPRC\$)	8.3.8
Value (VAL)	8.3.9
Verify File (FTYPE)	8.5.7
GOSUB:	
Statement	7.6
Computed	7.7
GOTO Statement	7.2
IF-THEN-ELSE Statement	7.4
IMAGE Statement	6.5.5
Immediate Execution Mode	4.2.2
Information Block, Parameter	11.6
INPUT Access Mode	6.3.5.2
Input:	
Errors	6.6.2.6
Function Key, Erase	4.8.9
Prompting	6.6.2.5
INPUT:	
Statement	6.6, 6.6.1, 6.6.3
Statement with:	
Relative Record File	6.6.3.2
Sequential File	6.6.3.1
With AT Option Statement	6.6.2.2
With BELL Option Statement	6.6.2.4
With SIZE Option Statement	6.6.2.3
Input/Output Statement	Section 6
Insert Character Function Key	4.8.7
Instruction, Loop	7.5
Integer:	
Data Format	11.7.1, F11-3
Field	6.5.5.2
Operands, Operator Logical with	Appendix F
Integer (INT) Function	8.2.5
INTEGER Statement	5.4.3.3
INTERNAL Format	6.3.2.2
INTERNAL Format Data, Memory Requirements for	T6-1
Internal Subprogram	9.3
Intrinsic:	
Function	Section 8
BASIC	T8-1
Invader	F2-1
Key	6.2.3
KEY Clause	6.6.3.3

- Key Indexed File (See KIF) 6.3.1.3
- Key:
 - Primary 6.2.3
 - Secondary 6.2.3
 - Value 6.2.3
- Keyboard Differences E.2
- Keyed File 12.4
 - Creation 12.4.4
 - Data Base Buffer Creation 12.4.3
 - Example 12.4.7
 - Format 12.4.2
 - Organization 12.4.1
 - Package (See KFP) 12.4
- KFP:
 - Memory Management 12.4.5
 - Keyed File Data Base Buffer 12.4.5.2
 - Record Buffer 12.4.5.3
 - Subroutine Memory Requirements 12.4.5.1
 - Subroutines 12.4.6
 - KFP 12.4, 12.4.6
 - SORT 12.3
- KFCLOS 12.4.6.10
- KFCREA 12.4.6.3
- KFDELRL 12.4.6.9
- KFGET 12.4.6.8
- KFINIT 12.4.6.2
- KFOPEN 12.4.6.4
- KFPUT 12.4.6.5
- KFREAD 12.4.6.7
- KFWRITE 12.4.6.6
- KIF 6.2.3
 - Data Entry (INPUT Statement) 6.6.3.3
 - Example Program 4.4
 - OPEN Statement with 6.3.1.3
 - Output 6.5.6.3
 - Shared 6.9.2
 - With RESTORE Statement 6.7.3.2
- Length (LEN) Function 8.3.3
- LET Statement 5.5
- LIBRARY Statement 11.4
- Life, File 6.3.4
- Lines Command, DELETE 3.8.3
- Link Control File Example F11-1
- Linking, Assembly Language
 - Subroutine 11.3
- LIST Command 3.7, 4.4
- Listing Example, Object F11-6
- Literal Field 6.5.5.6
- Locked, Record 6.9
- Logical AND Operator 5.9.4.3, F.3
- Logical, Expression 5.11.2
- Logical NOT Operator F.4
- Logical Operator 5.9.4
- Logical OR Operator F.2
- Loop Instruction 7.5
- MAIN Command, EDIT 3.6.1
- Match String (SPAN) Function 8.3.7
- Mathematical Function 8.2
- Memory Requirements for INTERNAL
 - Format Data T6-1
- MERGE Command 3.10, 4.7
- Messages, Error Appendix D
- Mode:
 - Immediate Execution 4.2.2
 - Program Development 4.2.1
 - Multiple, Statement 4.2.2
- Natural Logarithm (LOG) Function 8.2.6
- NEW Command 2.2.2, 3.2, 6.4
- NEXT Statement 7.5, 7.8
- NUM Command 2.2.2, 3.3
- Numeric:
 - Array Accessing 11.8.1
 - Constant 5.3.1
 - Data 5.2
 - Function 8.3.4
 - Names, Variable 5.4.2
 - Types 5.4.3
 - Variable 5.4.1
- Object Listing Cross Reference
 - Example F11-8
- Object Listing Example F11-6
- OLD Command 3.5, 4.4, 4.5, 6.4
- ON ERROR Statement 7.8
- ON-GOTO Statement 7.3
- OPEN:
 - Statement With:
 - KIF 6.3.1.3
 - Relative Record File 6.3.1.2
 - Sequential File 6.3.1.1
- Operating:
 - System:
 - DNOS E.1
 - DXM E.1
 - DX10 E.1
 - Pathnames E.3.3
- Operations Initial Section 2
- Operator 5.1, 5.9
 - Arithmetic 5.9.1, T5-1
 - Logical 5.9.4
 - Logical AND 5.9.4.3, F.3
 - Logical NOT F.4
 - Logical OR F.2
 - Logical with Integer Operands Appendix F
 - Priority 5.10
 - Relational 5.9.3, T5-2
 - String 5.9.2
- Option, AT 6.5.3.2
- OPTION BASE Statement 5.6.2
- Option:
 - BELL 6.5.3.4
 - ERASE ALL 6.5.3.1
 - SIZE 6.5.3.3
- OPTION Statement 4.9
- Option, USING 6.5.4
- OPTION 1 Statement 4.9.1
- OPTION 2 Statement 4.9.2

- Options, Output 6.5.3
- OUTPUT Access Mode 6.3.5.1
- Output:
 - Device 6.5.1
 - File 6.5.6
 - KIF 6.5.6.3
 - Options 6.5.3
 - Relative Record File 6.5.6.2
 - Sequential File 6.5.6.1
- Parameter:
 - Data Values 11.7
 - Information Block 11.6
- Pathnames:
 - DNOS E.3.3.1
 - DXM E.3.3.2
 - DX10 E.3.3.1
 - Operating System E.3.3
- Physical Record Length 6.3.3.3
- Position (POS) Function 8.3.5
- Preceding Line Function Key,
 - Display 4.8.5
- Primary, Key 6.2.3
- PRINT Statement 6.5
- Priority, Operator 5.10
- Procedure Shared E.5
- Procedures, User-Defined Section 9
- Program Development Mode 4.2.1
- Program Development 1.4
- Program:
 - Edit 4.4
 - Example Appendix H
 - Size Restriction, DXM E.4
- Prompt 4.2
- Prompting, Input 6.6.2.5
- PUNCTUATION Statement 6.8

- Random Number (RND) Function 8.5.1
- RANDOMIZE Statement 8.5.2
- READ Statement 6.7.2
- Real Data Format 11.7.2, F11-4
- REAL Statement 5.4.3.1
- Record Length:
 - File 6.3.3
 - Fixed 6.3.3.2
 - Physical 6.3.3.3
 - Variable 6.3.3.1
- Record Locked 6.9
- Recursive Function 9.2.3
- Relational:
 - Expression 5.11.4
 - Operator 5.9.3, T5-2
- Relative Record:
 - File 6.2.2
 - INPUT Statement with 6.6.3.2
 - OPEN Statement with 6.3.1.2
 - Output 6.5.6.2
- Remark (REM) Statement 4.3.1
- RENAME Command 3.9
- Repeat Function Key 4.8.12
- Repeat (RPT\$) Function 8.3.6
- Replay Function Key 4.8.13
- REPRINT Statement 6.5.7
- RESEQUENCE Command 3.4, 4.5
- Reset Cursor Function Key 4.8.18
- RESTORE:
 - Statement 6.7.3
 - KIF with 6.7.3.2
- Restrictions, DXM File E.3.2
- Resume Execution Function Key 4.8.16, 10.3
- Return Character (INKEY\$) Function 8.5.5
- Return Function Key 4.8.4
- RETURN Statement 7.6
- Return With EOF Function Key 4.8.20
- RUN Command 2.2.4, 3.13, 6.4

- SAVE Command 2.2.2, 3.12
- SCRATCH Statement 6.5.8
- Secondary, Key 6.2.3
- Segment (SEG\$) Function 8.3.11
- Semicolon Data Separator 6.5.2.2
- Sequential:
 - File 6.2.1
 - INPUT Statement with 6.6.3.1
 - OPEN Statement with 6.3.1.1
 - Output 6.5.6.1
- Shared:
 - File 6.9
 - KIF 6.9.2
- Sign (SGN) Function 8.2.7
- Sine (SIN) Function 8.2.8
- SIZE Option 6.5.3.3
- Size Restriction, DXM Program E.4
- Source Code Example F11-6
- Space Forward Function Key 4.8.1
- Square Root (SQR) Function 8.2.9
- Statement:
 - ACCEPT 6.6, 6.6.4
 - ASSIGN 5.7.1
 - CALL 9.3.1, 11.5
 - CLOSE 5.7.2, 6.4
 - Computed GOSUB 7.7
 - Control Section 7
 - DATA 6.7.1
 - DECIMAL 5.4.3.2
 - Define (DEF) 9.2, 9.2.1
 - DIM 5.6.1
 - DISPLAY 6.5
 - END 7.9
 - ESUB 9.3.2.2
 - FOR 7.5
 - Function End (FNEND) 9.2, 9.2.2
 - GOSUB 7.6
 - GOTO 7.2
 - IF-THEN-ELSE 7.4
 - IMAGE 6.5.5
 - INPUT 6.6, 6.6.1, 6.6.3
 - With AT Option 6.6.2.2
 - With BELL Option 6.6.2.4
 - With SIZE Option 6.6.2.3

- Input/Output Section 6
- INTEGER 5.4.3.3
- KIF With RESTORE 6.7.3.2
- LET 5.5
- LIBRARY 11.4
- Multiple 4.2.2
- NEXT 7.5, 7.8
- ON ERROR 7.8
- ON-GOTO 7.3
- OPTION 4.9
- OPTION BASE 5.6.2
- OPTION 1 4.9.1
- OPTION 2 4.9.2
- PRINT 6.5
- PUNCTUATION 6.8
- RANDOMIZE 8.5.2
- READ 6.7.2
- REAL 5.4.3.1
- Remark (REM) 4.3.1
- REPRINT 6.5.7
- RESTORE 6.7.3
- RETURN 7.6
- SCRATCH 6.5.8
- STEP 7.5
- STOP 7.10
- SUB 9.3.2.1, 9.3.2.2
- SUBEND 9.3.2.4
- SUBEXIT 9.3.2.3
- TO 7.5
- Variables in:
 - CALL 9.5
 - DEF 9.5
- Statement With:
 - KIF OPEN 6.3.1.3
 - Relative Record:
 - File, INPUT 6.6.3.2
 - File OPEN 6.3.1.2
 - Sequential:
 - File, INPUT 6.6.3.1
 - File OPEN 6.3.1.1
- Step by Step Execution 10.4
- Step Function Key 4.8.17, 10.4
- STEP Statement 7.5
- STOP Statement 7.10
- String:
 - Array Accessing 11.8.2
 - Constant 5.3.2
 - Data Format 11.7.4, F11-5
 - Expression 5.11.3
 - Function 8.3
 - Operator 5.9.2
 - Variable 5.4.4
 - String (STR\$) Function 8.3.12
- SUB Statement 9.3.2.1, 9.3.2.2
- SUBEND Statement 9.3.2.4
- SUBEXIT Statement 9.3.2.3
- Subprogram:
 - External 9.3
 - Internal 9.3
- Subroutine:
 - Assembly Language Section 11
 - Creation, Assembly Language 11.2
 - Linking, Assembly Language 11.3
- Succeeding Line Function Key, Display 4.8.6
- Suspend Execution Function Key 4.8.19
- Syntax Diagrams Appendix G
- System:
 - Differences Appendix E
 - DNOS Operating E.1
 - DXM Operating E.1
 - DX10 Operating E.1
 - Pathnames, Operating E.3.3
- Tab Function Key 4.8.11
- Tab (TAB) Function 8.5.8
- Table, Truth T5-3
- Tangent (TAN) Function 8.2.10
- Test for Duplicate Keys (DUP) Function 8.5.10
- Time (TIME\$) Function 8.4.2
- TO Statement 7.5
- TRACE Command 10.5
- Truth Table T5-3
- Types, Numeric 5.4.3
- UNBRKPNT Command 10.2.1
- UNLOCK Command 6.9
- UNTRACE Command 10.5
- UPDATE:
 - Access Mode 6.3.5.3
 - Command 3.11
- Uppercase (UPRC\$) Function 8.3.8
- User-Defined Procedures Section 9
- USING Option 6.5.4
- Value, Key 6.2.3
- Value (VAL) Function 8.3.9
- Values Associated With, Function Key T4-1
- Variable 5.4
 - Numeric 5.4.1
 - Numeric Names 5.4.2
 - Record Length 6.3.3.1
 - String 5.4.4
- Variables in:
 - CALL Statement 9.5
 - DEF Statement 9.5
- Verify File (FTYPE) Function 8.5.7
- Virtual, Array 5.7

USER'S RESPONSE SHEET

Manual Title: TI BASIC Reference Manual (2308769-9701)

Manual Date: 1 December 1983 Date of This Letter: _____

User's Name: _____ Telephone: _____

Company: _____ Office/Department: _____

Street Address: _____

City/State/Zip Code: _____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

CUT ALONG LINE

Location in Manual	Comment/Suggestion
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

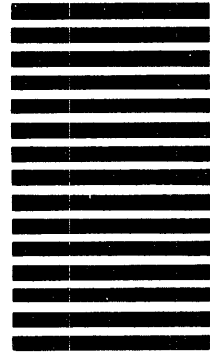
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS GROUP

ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769



FOLD