



HP 2000 Computer System

HP 2000 BASIC

Reference Manual

Part No. 22687-90001

Printed in USA 5/76

Update #3 Incorporated 11/77

Microfiche No. 22687-90002

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HP 2000 BASIC

Reference Manual



HEWLETT-PACKARD COMPANY
5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA 95050

ACKNOWLEDGEMENT

Hewlett-Packard wishes to acknowledge the substantial contribution to the development of the 2000 Computer System made by members of the professional staff of the University Computer Center of the University of Iowa. HP feels that this has been an unusually productive and cordial relationship between an industrial firm and an institution of higher education.

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page. Within the manual, changes are marked with a vertical bar in the margin.

Pages	Effective Date	Pages	Effective Date
Title	May 1976	8-6 and 8-7.	May 1976
ii to xii	Apr 1977	8-8	Apr 1977
xiii	May 1976	8-9 and 8-10.	May 1976
1-0 to 1-13.	Apr 1977	9-1 to 9-17.	Apr 1977
2-1 and 2-2.	Apr 1977	9-18	May 1976
2-3 and 2-4.	May 1976	9-19 to 9-23	Apr 1977
2-5	Apr 1977	10-1 to 10-3	Apr 1977
2-6 and 2-7.	May 1976	10-4 to 10-6	Sep 1976
2-8	Apr 1977	10-7	May 1976
2-9 and 2-10.	May 1976	10-8	Apr 1977
2-11	Apr 1977	10-9	May 1976
2-12 to 2-25.	May 1976	10-10 and 10-11	Apr 1977
2-26 and 2-27.	Apr 1977	10-12 and 10-13	May 1976
2-28 to 2-30.	May 1976	10-14	Sep 1976
2-31 and 2-32.	Apr 1977	10-15 to 10-18	May 1976
2-33 to 2-36.	May 1976	10-19	Apr 1977
3-1 to 3-10.	May 1976	10-20	May 1976
3-11	Jul 1976	10-21	Apr 1977
3-12 to 3-16.	May 1976	10-22	May 1976
4-1 and 4-2.	May 1976	10-23	Apr 1977
4-3	Apr 1977	10-24 to 10-30	May 1976
4-4 to 4-12.	May 1976	11-1 to 11-3	Apr 1977
5-1	Apr 1977	11-4	May 1976
5-2	Sep 1976	11-5	Apr 1977
5-3	May 1976	11-6	May 1976
5-4	Jul 1976	11-7 and 11-8.	Apr 1977
5-5 to 5-7.	May 1976	11-9	Jul 1976
5-8	Jul 1976	11-10 to 11-18	May 1976
5-9 to 5-10.	May 1976	11-19 to 11-20b	Apr 1977
5-11	Jul 1976	11-21 to 11-30	May 1976
5-12 to 5-14.	May 1976	11-31 and 11-32	Apr 1977
5-15	Apr 1977	11-33	Apr 1977
5-16	Jul 1976	11-34	May 1976
5-17 and 5-18.	Apr 1977	11-35 and 11-36	Jul 1976
5-19	Jul 1976	11-37 and 11-38	Apr 1977
5-20	May 1976	11-39	Sep 1976
5-21	Jul 1976	11-40 to 11-49	May 1976
5-22	May 1976	11-50	Apr 1977
5-23 to 5-25.	Jul 1976	11-51 and 11-52	May 1976
5-26	Sep 1976	11-53	Jul 1976
6-1 to 6-9.	May 1976	11-54 to 11-56	May 1976
7-1	Apr 1977	11-57	Apr 1977
7-2 to 7-4.	May 1976	11-58 to 11-60	May 1976
7-5	Apr 1977	11-61	Jul 1976
7-6	Sep 1976	11-62 to 11-63	May 1976
7-7 to 7-8.	May 1976	11-64	Jul 1976
8-1 to 8-4.	May 1976	11-65 to 11-73	May 1976
8-5	Apr 1977	11-74	Jul 1976

LIST OF EFFECTIVE PAGES (continued)

The List of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page. Within the manual, changes are marked with a vertical bar in the margin.

Pages	Effective Date	Pages	Effective Date
11-75 to 11-84	May 1976	D-1 to D-4	Apr 1977
11-85	Jul 1976	D-5 and D-6	May 1976
11-86 to 11-91	May 1976	D-7	Apr 1977
11-92	Apr 1977	E-1 and E-2	Apr 1977
11-93 to 11-96	May 1976	F-1 to F-10	Apr 1977
11-97 and 11-98	Apr 1977	G-1 to G-10	Sep 1976
11-99 and 11-100	May 1976	I-1 to I-7	Apr 1977
A-1 to A-5	Apr 1977		
B-1	May 1976		
C-1 to C-12	May 1976		
C-13	Apr 1977		

PRINTING HISTORY

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition	Sep 1975
Second Edition	May 1976
Update Number 1	Jul 1976
Update Number 2	Sep 1976
Update Number 3	Apr 1977
Update Number 3 Incorporated	Nov 1977

This publication is the user's reference manual for the HP 2000 Computer System. It provides information for logging on and developing and executing BASIC language programs. Included are instructions for using the system's Remote Job Entry facility. Other manuals which may provide useful information when using this manual are:

- *Learning Timeshare BASIC* (22687-90009) — An introduction to the BASIC language and a tutorial explanation of statements and commands.
- *HP 2000 Operators Manual* (22687-90005) — A guide to operating the 2000 Computer System on a daily basis including administrative procedures.
- *Pocket Guides* (22687-90003, 22687-90007) — Summaries of system capabilities for user's and the system operator.
- *TSP/2000-HASP User's Manual* (20240-90002) — A guide to using the Telecommunications Supervisory applications Package (TSP) with the Remote Job Entry facility.
- *TSP/2000-HASP Application Manager's Manual* (20240-90001) — A manager's guide to administering the Telecommunications Supervisory applications Package (TSP).

This manual is organized into eleven sections, six appendices, and an index.

- Section I — Introducing 2000/Access BASIC. This section is a description of the system, its capabilities, and how to use it.
- Section II — Introduction to BASIC Programming. This section describes key elements of programming in BASIC on the 2000 Computer System.
- Section III — Programming With Arrays. This section describes statements and techniques for using arrays in programs.
- Section IV — Programming With Strings. This section describes statements and techniques for using strings in programs.
- Section V — Files. This section describes the types of files used on the system, file read and write operation, and techniques of file organization.
- Section VI — Formatted Output. This section describes techniques of creating formatted output for applications such as report generation.
- Section VII — System Facilities. This section describes techniques for linking programs and passing data from one program to another.
- Section VIII — Security and the Library Hierarchy. This section describes the levels of program and file security available on the system. Techniques for using the security and library structure are discussed.

Preface

- Section IX — Using the Remote Job Entry Facility. This section describes the Remote Job Entry capability of the 2000 Computer System and provides instructions for its use.
- Section X — Commands. This section provides the syntax and definition of all the user commands available.
- Section XI — BASIC Language Reference. This section provides a definition of the BASIC Language. A rigorous definition of the terminology precedes the syntax and definition of the BASIC statements. Experienced programmers can use this section as a detailed reference for the BASIC language as implemented on the 2000 system. Less experienced programmers should first read the introductory sections of the manual, and if you are a beginning programmer, *Learning Timeshare BASIC* is recommended.
- Appendix A — Using the ASCII Character Set. This appendix provides a description of the ASCII character set and its use in BASIC language statements.
- Appendix B — How to Prepare A Paper Tape Off-Line. This appendix describes the preparation of paper tapes for input to the system.
- Appendix C — Error Messages. This appendix lists the error messages that you could receive when using the system.
- Appendix D — Terminal Interface. This appendix provides some of the operating characteristics of terminals used with the system.
- Appendix E — Additional Library Features. This appendix describes some of the special functions available through the system operator.
- Appendix F — Formal Syntax for 2000 BASIC. This appendix presents the BASIC language syntax in Backus Naur form (BNF).

The text in this manual is primarily for reference and has been written as a definition rather than an explanation of the BASIC language. Experienced programmers can use the detailed reference material given in Sections X and XI to look up the syntax of commands and statements. Less experienced programmers should read the entire manual.

Simple programming examples are used throughout the manual. The examples have been selected to demonstrate how the language is used and are not intended to be examples of efficient programming

CONTENTS

Section I	Page		
INTRODUCING 2000 BASIC			
What is the 2000 System?	1-1	GOTO Statement	2-13
BASIC LANGUAGE	1-1	END/STOP Statements	2-14
Operating System Software	1-1	IF Statement	2-14
System Hardware	1-1	FOR/NEXT Statement	2-15
System Resources	1-3	READ/DATA/RESTORE Statements	2-17
Input/Output Devices	1-3	INPUT Statement	2-20
Remote Control of HPIB Devices	1-3	ENTER Statement	2-21
File Supervision	1-3	GOSUB/RETURN Statements	2-22
Security	1-3	DEF Statement	2-25
Remote Job Entry (RJE)	1-3	Commands	2-26
2000 to 2000 Communications	1-3	Your Work Space	2-26
Terminal Time	1-3	Using Your Work Space	2-26
What Does the System Do?	1-4	Your Library	2-30
How Do You Use the System?	1-6	Accessing Your Library	2-30
Account and Library System	1-6	Library Catalog	2-31
System Master Account	1-6	Paper Tape	2-33
Group Master Accounts	1-6	Punching the Tape	2-33
Individual Accounts	1-6	Reading the Tape	2-34
Dedicated Application Environment	1-7	Magnetic Tape Cartridges	2-35
Operating the Equipment	1-7	Writing to the Cartridge	2-35
Connecting to the System	1-7	Reading from the Cartridge	2-36
LINE/LOCAL SWITCH	1-8		
FULL DUPLEX/HALF DUPLEX SWITCH	1-8	Section III	Page
AUTO LF Key	1-9	PROGRAMMING WITH ARRAYS	
Terminal Speed	1-9	What Are Arrays?	3-1
Logging On and Off the System	1-9	Referencing Arrays	3-1
HELLO Command	1-9	Referencing Array Elements	3-1
Errors During Logging On	1-10	Dimensioning Arrays	3-2
ECHO Command	1-11	Placing Values into Arrays	3-3
TIME Command	1-12	Printing Data from Arrays	3-4
MESSAGE Command	1-12	Redimensioning Arrays	3-6
BYE Command	1-12	Initializing Arrays	3-7
You and the System Operator	1-12	Array Operations	3-9
		Array Addition/Subtraction	3-9
		Array Multiplication	3-11
		Array Inversion	3-14
		Array Transposition	3-15
		Array Scalar Multiplication	3-16
Section II	Page	Section IV	Page
INTRODUCTION TO BASIC PROGRAMMING		PROGRAMMING WITH STRINGS	
Constants, Variables, and Expressions	2-1	What Are Strings?	4-1
Constants	2-1	String Character Set	4-1
Numeric Constants	2-1	Numeric Equivalents of Characters	
String Constants	2-2	(An Alternate Form)	4-2
Logical Constants	2-3	Upper and Lower Case Letters	4-2
Variables	2-3	How Do You Reference Strings?	4-3
Subscripted Variables	2-4	Naming Strings	4-3
Expressions	2-4	Dimensioning Strings	4-3
Operators	2-5	Substrings	4-4
Evaluating Expressions	2-6	How Do You Use Strings?	4-4
Functions	2-8	Placing Values in Strings	4-5
Statements	2-9	Simple Assignment	4-5
ASSIGNMENT Statement	2-9	String Data	4-5
PRINT Statement	2-10	Setting Strings Equal to String Valued	
Printing Numeric Items	2-11	Functions	4-6
Printing String Items	2-11	Using Strings in Relational Operations	4-7
Print Functions	2-11		
REM Statement	2-13		

CONTENTS (continued)

String Statements and Functions	4-7	Account Accessing Capabilities	8-5
String Valued Statements and Functions	4-7	Program and File Restrictions	8-6
Numeric Valued Statements and Functions	4-9	Program States	8-6
Printing Strings	4-11	File States	8-7
		Controlling Simultaneous Writing on Files	8-8
Section V	Page	Section IX	Page
FILES		DATA COMMUNICATIONS	
What Are Files?	5-1	What is Remote Job Entry?	9-1
BASIC Formatted Files	5-2	What Host Systems Can You Communicate With?	9-3
Creating BASIC Formatted Files	5-3	Multileaving RJE Workstation (MRJE/WS)	9-3
Opening and Closing a BASIC Formatted File	5-5	2770/2780/3780 Terminal	9-4
Accessing BASIC Formatted Files	5-6	User 200 Terminal	9-4
Serial File Access	5-6	How Does RJE Work?	9-4
Direct File Access	5-12	How Do You Use Remote Job Entry?	9-6
ASCII Files	5-17	Sending Jobs Through the Card Reader	9-6
Creating and Purging ASCII Files	5-17	Retrieving Output on the Line Printer	9-7
Opening ASCII Files	5-20	Sending Jobs and Retrieving Output Through a Program	9-7
Printing to ASCII Files	5-20	How to Get <i>Your</i> Output	9-11
Using CTL Function with ASCII Files	5-22	Communicating with a Host System Through a Program	9-14
Using *Out=filename* with ASCII Files	5-23	Controlling Multiple Devices	9-15
Reading From an ASCII File	5-23	Remote Command Summary	9-17
		2000 to 2000 Communications	9-20
Section VI	Page	Transmitting Data Between Systems	9-21
FORMATTED OUTPUT		Transmitting Data Between Programs	9-22
What is Formatted Output?	6-1	2780 to 2780 Communication	9-23
How Do You Indicate Formatted Output?	6-1		
Using List	6-2	Section X	
Format String	6-3	COMMANDS	
Numeric Formatting Characters	6-3	What is a Command?	10-1
String Formatting Characters	6-3	Terms Used in this Section	10-1
Repetition Factors	6-3	ASCII File	10-2
Delimiters	6-5	Line Printer	10-2
Carriage Control Characters	6-5	Paper Tape Reader	10-2
String Constants	6-6	Paper Tape Punch	10-3
Using Formatted Output	6-7	Card Reader	10-3
		Card Reader/Punch/Interpreter	10-3
Section VII	Page	Link Terminal	10-4
SYSTEM FACILITIES		Magnetic Tape	10-4
Linking Programs	7-1	BASIC Formatted File	10-5
Passing Parameters	7-3	Block	10-5
Programmatic Error Detection	7-4	Device Designator	10-5
Programmatic Detection of Absence of Data	7-5	End-of-File Mark (EOF)	10-6
Programmatic Detection of Break	7-5	File Length	10-6
Programmatic Determination of Terminal Type	7-6	File Name	10-6
Executing Program Commands	7-6	Full Duplex	10-6
		General Device Designator	10-7
Section VIII	Page	Group Library	10-7
SECURITY AND THE LIBRARY HIERARCHY		Half Duplex	10-7
User Idcode Organization	8-1	Idcode	10-7
Private Library — Private User	8-3	Job Function Designator	10-8
Group Library — Group Master	8-3	Library	10-8
System Library — System Master	8-3		

CONTENTS (continued)

Library Name	10-8	File Name	11-5
Non-Sharable Device	10-8	File Number	11-5
OUT= FILE NAME	10-8	Function	11-5
Program Name	10-9	Function Reference	11-6
Program Reference	10-9	Literal String	11-7
Record	10-9	Logical Length	11-7
Record Length	10-9	Logical Size	11-7
Specific Device Designator	10-9	New Dimensions	11-8
Statement Number	10-9	Number	11-8
System Library	10-10	Numeric Constant	11-8
Work Space	10-10	Numeric Expression	11-9
Command Descriptions	10-10	Numeric Simple Variable	11-12
APPEND Command	10-11	Numeric Variable	11-12
BYE Command	10-11	Physical Length	11-12
CATALOG, GROUP, and LIBRARY		Physical Size	11-12
Commands	10-12	Primary	11-13
CREATE Command	10-13	Program Designator	11-13
CSAVE Command	10-13	Program Name	11-13
DELETE Command	10-13	Record Number	11-13
DEVICE Command	10-14	Relational Operator	11-13
ECHO Command	10-15	Return Variable	11-14
EXECUTE Command	10-15	Source String	11-14
FILE Command	10-16	Statement Number	11-15
GET Command	10-17	String	11-16
GROUP Command	10-17	String Expression	11-16
HELLO Command	10-18	String Length	11-16
KEY Command	10-19	String Simple Variable	11-17
LENGTH Command	10-19	String Value	11-17
LIBRARY Command	10-19	String Variable	11-17
LIST Command	10-20	Subscripted Variable	11-17
LOAD Command	10-20	Subscripted Designator	11-18
LOCK Command	10-21	Statements and Functions	11-19
MESSAGE Command	10-21	ABS Function	11-20
MWA Command	10-22	ADVANCE Statement	11-20
NAME Command	10-22	ASSIGN Statement	11-20a
PAUSE Command	10-23	ATN Function	11-23
PRIVATE Command	10-23	BRK Function	11-23
PROTECT Command	10-24	Break Capability	11-23
PUNCH Command	10-24	Values Returned by BRK Function	11-24
PURGE Command	10-25	CHAIN Statement	11-25
RENUMBER Command	10-26	CHR\$ Function	11-27
RUN Command	10-27	COM Statement	11-28
SAVE and CSAVE Commands	10-27	CON Function	11-28
SCRATCH Command	10-28	CONVERT Statement	11-29
SWA Command	10-28	Numeric to String	11-29
TAPE Command	10-29	String to Numeric	11-29
TIME Command	10-29	COS Function	11-29
UNRESTRICT Command	10-30	CREATE Statement	11-30
		CTL Function	11-31
		Line Printer	11-34
		Magnetic Tape	11-34
		Paper Tape	11-34
		Card Reader	11-35
		Reader/Punch/Interpreter	11-36
		Link Terminal	11-37
		RJE ASCII Files	11-38
		DATA Statement	11-38
		DEF Statement	11-38
		DIM Statement	11-39
Section XI	Page		
BASIC LANGUAGE REFERENCE			
Introduction	11-1		
BASIC Language Terms	11-1		
Array	11-1		
Array Element	11-2		
Array Name	11-2		
Character	11-3		
Constant	11-3		
Destination String	11-3		

CONTENTS (continued)

END Statement	11-39	PRINT # USING Statement	11-82
ENTER Statement	11-40	PURGE Statement	11-83
EXP Function	11-41	READ Statement	11-84
FILES Statement	11-41	READ # Statement	11-85
FOR and NEXT Statements	11-43	ASCII File Read Operations	11-87
Nesting For . . . Next Loops	11-44	REC Function	11-88
GOSUB and RETURN Statements	11-45	REM Statement	11-88
Multibranching	11-46	RESTORE Statement	11-89
Nesting GOSUB Statements	11-46	RND Function	11-90
GO TO Statement	11-47	SGN Function	11-91
IDN Function	11-47	SIN Function	11-91
IF . . . THEN Statement	11-48	SPA Function	11-92
IF END Statement	11-49	SQR Function	11-92
IF ERROR Statement	11-50	STOP Statement	11-92
IMAGE Statement	11-50	SYS Function	11-93
INPUT Statement	11-51	SYSTEM Statement	11-94
INT Function	11-52	TAB Function	11-95
INV Function	11-52	TAN Function	11-95
ITM Function	11-53	TIM Function	11-96
LEN Function	11-53	TRN Function	11-96
LET Statement	11-54	TYP Function	11-97
LIN Function	11-55	UNLOCK Statement	11-98
LINPUT Statement	11-56	UPDATE Statement	11-99
LINPUT # Statement	11-56	UPS\$ Function	11-100
LOCK Statement	11-57	ZER Function	11-100
LOG Function	11-58		
MAT Addition and Subtraction Statements	11-58	Appendix A	Page
MAT Assignment Statement	11-59	USING THE HP 2000 CHARACTER SET	A-1
MAT . . . CON Statement	11-59		
MAT . . . IDN Statement	11-59	Appendix B	Page
MAT INPUT Statement	11-60	HOW TO PREPARE A PAPER TAPE OFF-LINE	B-1
MAT . . . INV Statement	11-60		
MAT Multiplication Statement	11-61	Appendix C	Page
MAT PRINT Statement	11-62	ERROR MESSAGES	C-1
MAT PRINT # Statement	11-64	User Command Error Messages	C-1
MAT PRINT USING Statement	11-65	APPEND	C-1
MAT PRINT # USING Statement	11-65	CREATE	C-1
MAT READ Statement	11-66	CSAVE	C-1
MAT READ # Statement	11-66	DELETE	C-2
MAT Scalar Multiplication Statement	11-67	EXECUTE	C-2
MAT . . . TRN Statement	11-67	FILE	C-2
MAT . . . ZER Statement	11-68	GET	C-2
NEXT Statement	11-69	HELLO	C-2
NUM Function	11-69	LIST	C-2
POS Function	11-69	LOAD	C-3
PRINT Statement	11-70	LOCK	C-3
Numeric Output Formats	11-72	MESSAGE	C-3
Integers	11-72	MWA	C-3
Fixed-Point Numbers and All Other		NAME	C-3
Integers	11-73	PRIVATE	C-3
All Other Numbers	11-73	PROTECT	C-3
PRINT # Statement	11-74	PUNCH	C-3
Basic Formatted File Prints	11-74	PURGE	C-3
ASCII File Prints	11-76	RENUMBER	C-4
PRINT USING Statement	11-77	RUN	C-4
Carriage Control Characters	11-77	SAVE	C-4
Format Characters	11-77	SWA	C-4
Delimiters	11-78	UNRESTRICT	C-4
Groups	11-78	Language Processor Error Messages	C-5
Repetition Factors (Replicators)	11-78	Syntax Errors	C-5
Execution of the Print Using Statement	11-78	Execution Errors	C-6
Format Specifications	11-78		

CONTENTS (continued)

Compile Time Errors	C-6	Load	E-2
Programming Errors	C-7	Restore	E-2
Format Errors	C-11	Dump	E-2
Execution Warnings	C-12		
ASCII File Errors	C-13		
		Appendix F	Page
Appendix D	Page	FORMAL SYNTAX FOR 2000	
TERMINAL INTERFACE	D-1	BASIC	F-1
IBM 2741 Communications Terminal			
Interface	D-6	Appendix G	
		PROGRAMMING THE LINK TERMINAL	G-1
Appendix E	Page	Controlling the Indicator Lights	G-1
ADDITIONAL LIBRARY FEATURES	E-1	Using the Special Function Keys	G-1
Bestow	E-1	Controlling the Display Unit	G-1
Copy	E-1	Controlling Multiple Link Terminals	
		through a single Program	G-1
		Interfacing with devices on the local HPIB	G-1

ILLUSTRATIONS

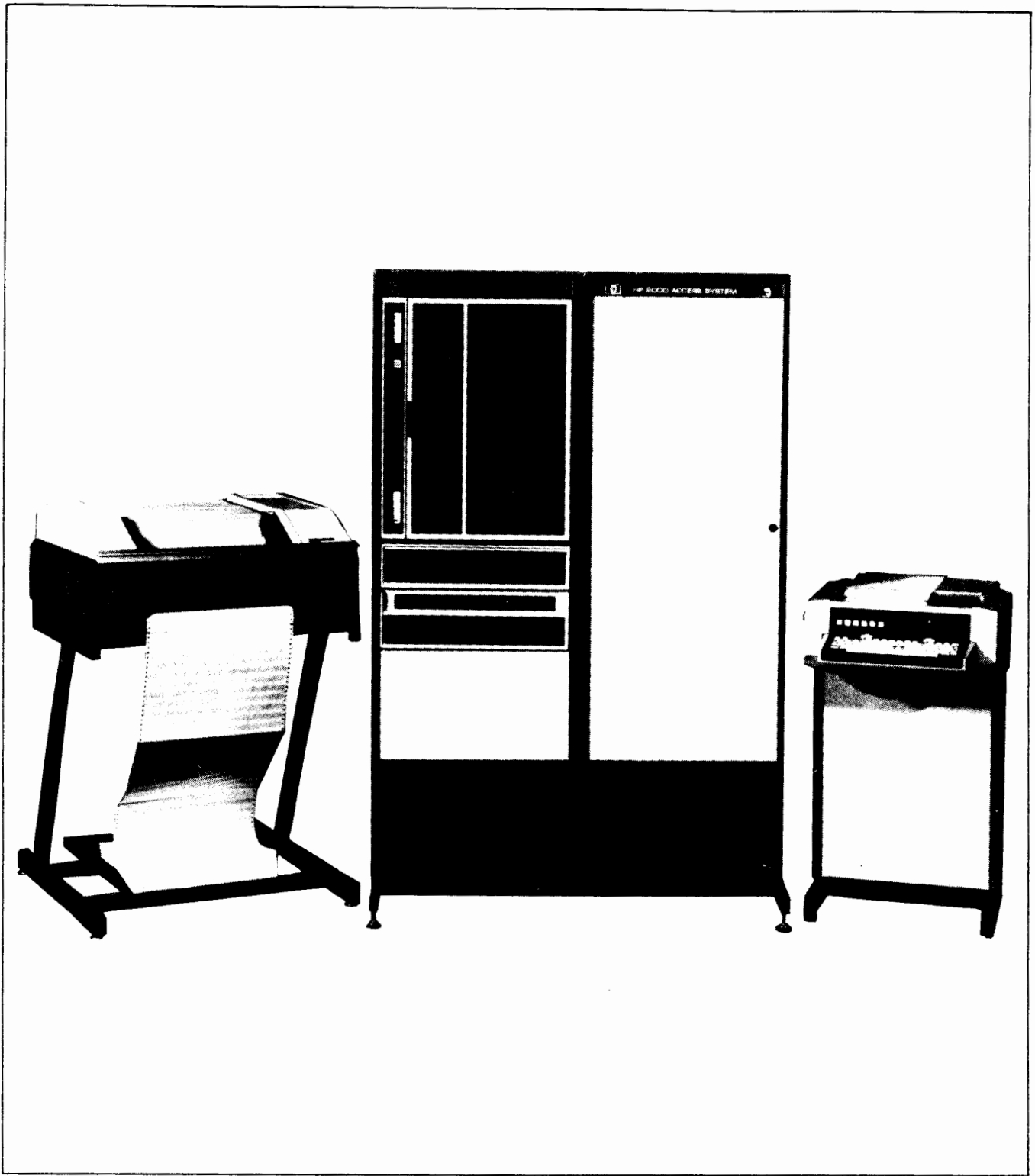
Figure	Page		
Typical HP 2000 Computer System	1-0	Idcode/Group Account Structure	8-2
System Block Diagram	1-2	Program Access States	8-6
Example of Program Access and Execution	1-5	File Access States	8-7
Sample Account Structure	1-7	Elements of an HP 2000 Data Communications Facility	9-2
BASIC Formatted and ASCII File Devices	5-2	Typical RJE Configuration For An IBM Host System	9-5
Record Format in BASIC Formatted Files	5-3	Example of an RJE Job Deck	9-7
Sample File Organization	5-5	Example of Forms Assignment to Route Output	9-13
Serial File Organization	5-6	The HPIB Structure of the Link Terminal	G-1
Direct File Organization	5-6		

TABLES

Table	Page		
Remote Terminal Connection Procedures	1-8	Terminal Configurations of 2770/2780/3780 Emulator	9-23
Selected System Operator Commands	1-13	User Commands	10-10
Summary of File Statements and Functions	5-16	Statements	11-19
BASIC Formatted Versus ASCII File Usage	5-26	Functions	11-19
Account Accessing Capabilities	8-5	ASCII Character Set	A-2
RJE Compatible Host Systems	9-3	2000 Access ASCII/EBCDIC Character Set	A-3
IBM MRJE Workstation Host Functions	9-3	Differences Between HP 2000, IBM and CDC Graphics	A-5
Job Function Designators	9-4	Matrix of Terminal Types	D-3
Summary of 360 HASP Remote Commands	9-17	IBM 2741 ASCII Character Simulation	D-7
Summary of Some IBM ASP Remote Commands	9-17	Repertoire of HPIB Messages	G-8
Summary of CDC EXPORT/IMPORT Remote Commands	9-18		
Some Useful Host System Manuals	9-19		
Communication Between Local and Remote 2000 Host Functions	9-20		

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
[]	An element inside brackets is <i>optional</i> . Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example: $\begin{bmatrix} A \\ B \end{bmatrix}$ user may select A or B or neither
{ }	When several elements are stacked within braces the user must select one of these elements. Example: $\left\{ \begin{array}{l} A \\ B \\ C \end{array} \right\}$ user must select A or B or C.
italics	Lowercase italics denote a parameter which must be replaced by a user-supplied variable. Example: CALL <i>name</i> <i>name</i> one to 15 alphanumeric characters.
underlining	Dialogue: Where it is necessary to distinguish user input from computer output, the input is underlined. Example: NEW NAME? <u>ALPHA1</u>
superscript C	Control characters are indicated by a superscript C Example: Y ^C
<i>return</i>	<i>return</i> in italics indicates a carriage return
<i>linefeed</i>	<i>linefeed</i> in italics indicates a linefeed
...	A horizontal ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted.



■ Figure 1-1. Typical HP 2000 Computer System

WHAT IS THE 2000 SYSTEM ?

The HP 2000 System is a terminal-oriented computer system using a powerful version of the BASIC language. Up to 32 users may access the system simultaneously through hardwired terminals or over ordinary telephone lines using modems. The system may also use a variety of peripherals such as card readers, paper tape punches, magnetic tape drives, paper tape readers, card reader/punch/interpreters, line printers, and serial link terminals. A typical system is shown in figure 1-1.

In addition to system users there is a system manager and a system operator. The system manager is responsible for establishing the initial system configuration and setting operating policies. The system operator is responsible for daily system operation and the granting of system resources such as storage space and terminal time. In the remainder of this manual there will be additional references to the system manager and system operator functions. Detailed descriptions of their duties are contained in the HP 2000 System Operator's Manual (22687-90005).

The remainder of this section describes briefly major system components, capabilities, and how to use the system. This manual assumes that you have programming experience in a language similar to BASIC and are familiar with the type of terminal used on your system.

The system is made up of three major components, the BASIC language, the operating system software, and the system hardware. The components allow you to make use of a variety of system resources.

BASIC LANGUAGE

Hewlett-Packard 2000 BASIC contains additional programming features beyond those in most versions of the BASIC language. These features, in the form of an extended set of statements, give you powerful tools for applications concerned with on-line data management and computation.

OPERATING SYSTEM SOFTWARE

The operating system software controls the overall system operation, supervises file activities, controls all input/output operations, and provides utility functions. System operation is discussed in detail in the HP 2000 System Operator's Manual (22687-90005).

SYSTEM HARDWARE

The system uses two computers, one for handling input/output operations (I/O processor) and another for executing BASIC programs (main processor). In addition it uses high speed disc drives for program and file storage, magnetic tape units for system backup and offline storage, and various terminals, printers, card readers, and other input/output devices. The peripheral devices available will vary depending on your system configuration. The organization of a

typical system is shown in figure 1-2. Information on individual system hardware units is contained in the installation, maintenance, and operator manuals for the specific device.

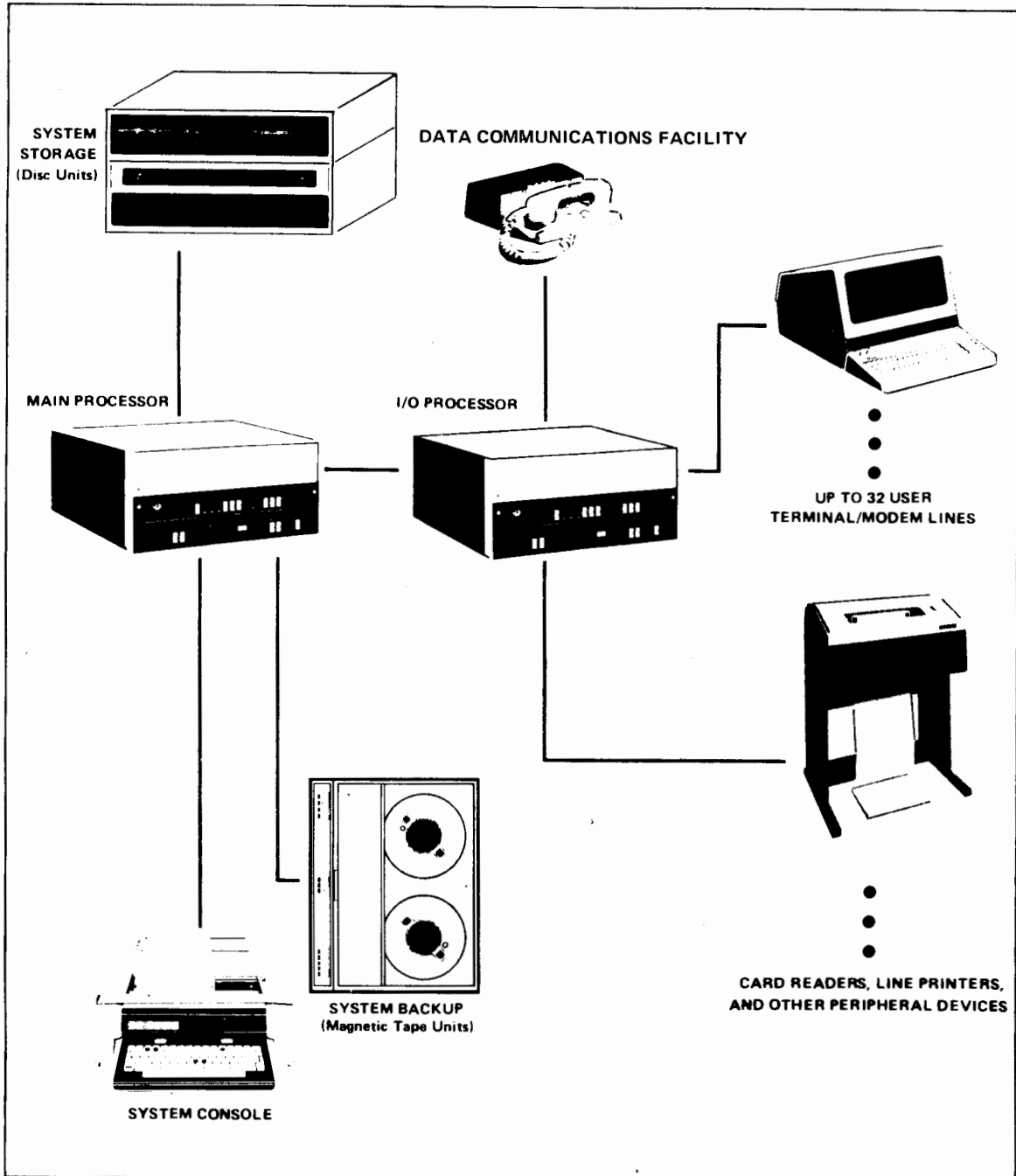


Figure 1-2. System Block Diagram

SYSTEM RESOURCES

The system resources consist of input/output devices, file supervision, terminal time, security, and remote job entry capabilities. These resources are available either directly through your terminal or on request from the system operator. The system operator accesses the system through the system console and performs privileged operations affecting system resources.

INPUT/OUTPUT DEVICES. Your terminal is the best example of a system input/output device. In addition to your terminal, a variety of input/output units are available for use as "non-sharable devices". These devices are peripherals such as line printers, magnetic tape drives and link terminals. They are non-sharable in the sense that once you begin using such a device it remains under your control and may not be used by others. The device is released by program or command termination. A device may also be assigned for exclusive use from time to time by the system operator to a specific user. Once assigned to you, a line printer, for example, may not be used by other users. Devices are deassigned by the system operator and may be immediately reassigned to another user.

REMOTE CONTROL OF HPIB DEVICES. A Hewlett-Packard Interface Bus (HPIB) can be controlled programmatically up to 1.2 miles from the system. Access to the HPIB is made possible through a link terminal to which up to 13 HPIB compatible devices may be connected locally. A complete discussion of how to interact with the HPIB is provided in Appendix G.

FILE SUPERVISION. The system controls access to files. You are given a limit to the amount of file space you may use at the time your account is opened by the system operator. Up to 65,535 blocks of file storage can be allocated to any one user account. This amount may be less depending on your system's configuration and operational policies. The amount of file space can be increased or decreased at any time by the system operator. This allows you to begin programming with a relatively small file space and increase it later as your storage needs grow.

SECURITY. When your account is opened by the system operator you will be assigned an account number and a password (security code). This password is used to limit the access to your account. Several people may share an account; or accounts may be assigned one to a user. Additional information on user accounts is contained in the description of the system account structure. Each program and file on the system is assigned one of four levels of security.

REMOTE JOB ENTRY (RJE). The system may be used to provide a remote work station for a larger computer system concurrent with other system operation. The RJE facilities are available directly through peripheral devices or from your program. You can then enter jobs, programs, or data in any language (FORTRAN, ALGOL, COBOL, etc.) available on the host system. A complete discussion of how to use the RJE capabilities of the system is contained in Section IX of this manual.

2000 TO 2000 COMMUNICATIONS. Using the RJE capabilities, data can be transferred between two HP 2000 systems. The data can be transferred between executing BASIC programs or directly between devices. 2000 to 2000 communication is discussed with the RJE facility in Section IX of this manual.

TERMINAL TIME. When your account is opened by the system operator you are given a limit to the total amount of terminal time you may use. Up to 65,535 minutes can be given. The system logs all terminal time used by your account. The time accrued may be reset to 0 by the system operator.

WHAT DOES THE SYSTEM DO?

The primary function of the system is to allow up to 32 concurrent users to develop and execute programs in the BASIC language. In addition the system provides you with resources for input/output, file maintenance, operator utility functions, system, program, and file security, and Remote Job Entry (RJE).

The BASIC language used on the system consists of statements for writing programs and commands for controlling both program execution and input/output operations. This manual assumes that you are familiar with a language similar to BASIC and that you know how to use the terminals available on your system.

The sample program in figure 1-3 illustrates some of the statements and commands available on the system for program access and execution. Program statements are numbered, commands are not. Special terminal keys are circled, and system responses are shaded.

Figure 1-3 illustrates logging on from a terminal, creating a data file, accessing, listing, and running a program. The program is already written and is stored in your account library. When run, the program asks you to enter string data at your terminal, and it then stores the entered data in a file. When you enter a slash (/), the program terminates.

OPERATION/STATEMENT	DESCRIPTION
<i>return</i> <i>linefeed</i>	Enter return/linefeed to get system attention
PLEASE LOG IN	System log on request
HELLO-B105,PASWRD <i>return</i>	Log on with your account number and password
(System Message)	Message varies with system
CREATE-FIL1,10 <i>return</i>	Create empty file FIL1 with 10 records
GET-PROG1 <i>return</i>	Get PROG1 from your library
LIST <i>return</i>	List program
PROG1	
10 REM BUILDS A LIST OF PROGRAM NAMES	Remark (not executed)
20 DIM A\$(80)	Define a string variable (to contain characters)
30 FILES FIL1	Specify file to be used
40 PRINT "TYPE / TO EXIT"	Output prompt messages
50 PRINT "ENTER PROGRAM NAME"	Output prompt messages
60 INPUT A\$	Accept input data
70 IF A\$="/" THEN 100	Test for exit and branch to END
80 PRINT #1; A\$	Write data sequentially to file
90 GOTO 50	Branch to statement 50
100 END	End of program
RUN <i>return</i>	Execute program
PROG1,	System prints program name
TYPE / TO EXIT	Program prompts
ENTER PROGRAM NAME	Program prompts
?TEST1 <i>return</i>	Enter program name
ENTER PROGRAM NAME	Program prompts
?TEST2 <i>return</i>	Enter next name
ENTER PROGRAM NAME	Program prompts
?/ <i>return</i>	Enter slash, to terminate
DONE	Program prompts
BYE <i>return</i>	Log off system
System Message	

Figure 1-3. Example of Program Access and Execution

HOW DO YOU USE THE SYSTEM?

The HP 2000 Computer System is designed to be extremely easy to use. It requires no complicated job control language; program entry is free field; and there is no separate compile operation. Using the system consists of obtaining an account number and password from the system operator, logging on, entering or retrieving a program, and running the program. In addition you can enter system commands to obtain a wide variety of information or cause the system to perform various activities for you.

ACCOUNT AND LIBRARY SYSTEM

The accounts used on the system are made up of three types, system, group, and user. Each type of account has slightly different capabilities and performs a different function in the typical system. Each account has its own library for storage of programs and files. The following paragraphs discuss briefly each of the account types and their capabilities.

SYSTEM MASTER ACCOUNT. The system master account is numbered A000 and is used only by the system master or system operator. This account is used to hold the system library containing programs and files that can normally be accessed by all system users. A listing of the accessible contents of the system library can be obtained using the LIBRARY command.

The A000 user can use the system operator commands such as DIRECTORY and REPORT, and the DUMP command. A complete discussion of system master capabilities is given in the HP 2000 System Operator's Manual (22687-90005).

GROUP MASTER ACCOUNTS. The group master or group librarian account has an account number made up of a letter and a digit followed by two zeros. For example, Z900, J100, and K700 are all group master accounts. A group account contains programs and files which may be accessed by any account within the group. For example, C100 is a group master account and all accounts from C100 through C199 are members of the group. The system master account (A000) is a special case of a group master account in which the system master is the group librarian of all accounts in the range A000 through A099.

INDIVIDUAL ACCOUNTS. The individual account has access to its own files and programs, its group library, and the system library. Note that the system library is the library belonging to account A000 and that each group library is the library belonging to the group master account. The individual account owner has control over his own library through the various commands and statements used to store, delete, or retrieve programs and files. In addition he may change the level of security for programs and files in his library.

Figure 1-4 illustrates the account structure of a small system. In this example, anyone logging on with individual account number B503 can access programs or files in his own library, in the group library B500, and the system library A000. He cannot access other individual libraries such as B597 or A093, or other group libraries such as B200 and C400. The group master who logs on with B500 can access only his own library and the system library. The system master (account A000) can access only his own library. This library is his group library and the system library as well as his own personal library.

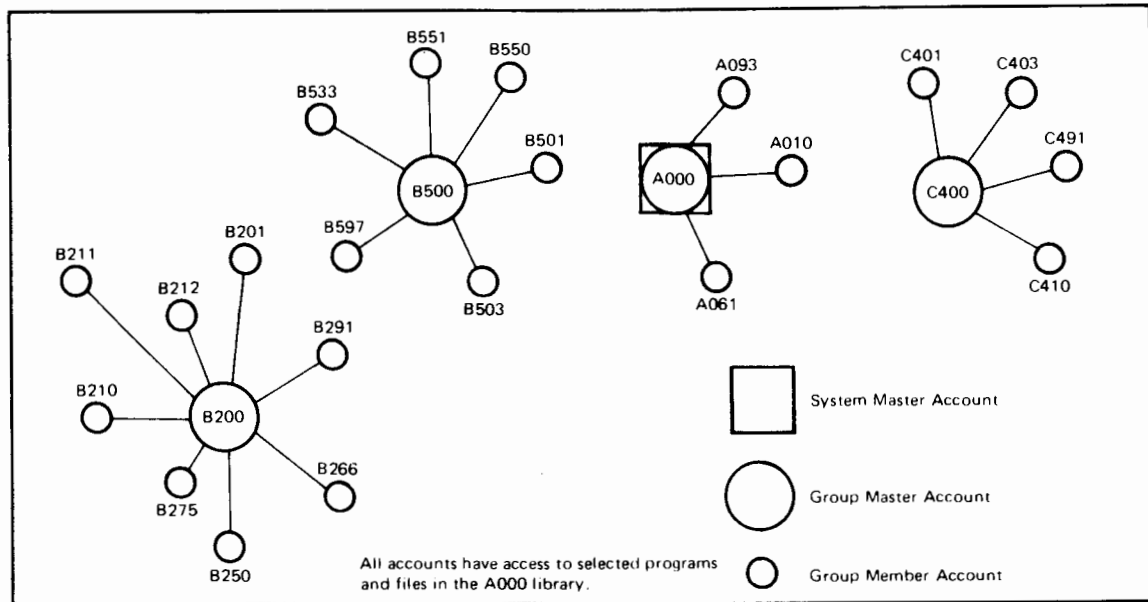


Figure 1-4. Sample Account Structure



DEDICATED APPLICATION ENVIRONMENT. An important function of the system master is to provide an environment in which any user with a particular account number can log on directly to an application program so that it appears to the user as if the system consisted solely of his own program. This is accomplished by the system master through the HELLO program and is made possible through the account accessing structure. Refer to Section VIII for a description of account accessing capabilities and the system master's use of the HELLO program to define a dedicated application environment.

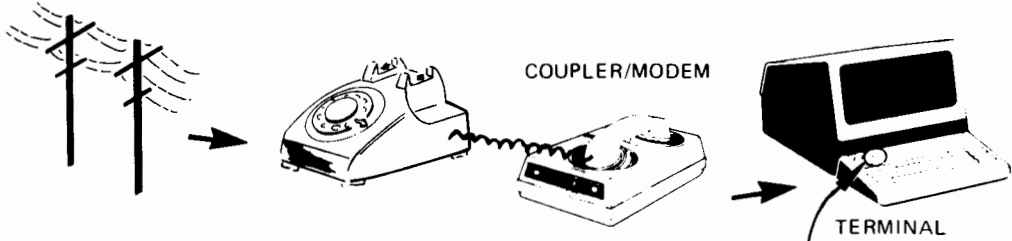
OPERATING THE EQUIPMENT

Several types of terminals can be used with the 2000 system. The terminal used must generate characters in ASCII code unless it is an IBM 2741. (Refer to Appendix D for a full discussion of terminal types.) It is not necessary for the terminal to be able to generate the entire ASCII character set, but if your terminal does not use the entire set, you may not be able to use the full capabilities of the system. For example, you can use all 128 ASCII characters but some terminals do not print the lower-case alphabet. The lower-case alphabet, while important in such applications as text editing and report generation, is not required for most analytical applications or for program development. An HP 7260A Optional Mark Reader can be connected in parallel with any of the supported terminals.

CONNECTING TO THE SYSTEM. To log on to the system, connection must be established between your terminal and the system. The way that this connection is made varies depending on the type of terminal used. If your terminal is wired directly to the system (hardwired) all that is required is to set the terminal mode to ON-LINE (or REMOTE) and the power switch to ON. If your terminal is remote from the system it must use a modem or Data Set to link your terminal to the system. Table 1-1 contains procedures for using most remote terminals.

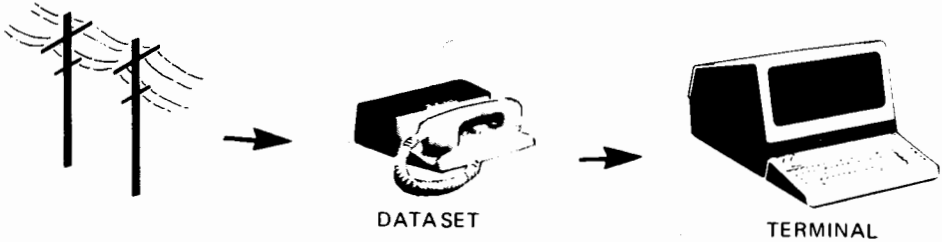
Table 1-1. Remote Terminal Connection Procedures

ACOUSTIC COUPLER MODEM AND TELEPHONE:



1. Set terminal mode to REMOTE and power switch to ON.
2. Set coupler power switch to ON.
3. If coupler has a duplex switch, set to FULL DUPLEX.
4. If coupler has a line switch, set to ON-LINE.
5. Remove telephone handset and dial the computer telephone number.
6. When the computer responds with a high pitched tone, place the handset into the coupler receptacle (the correct handset position should be marked on the coupler).
7. Type return linefeed and log on.

DATA SET



1. Set terminal mode to REMOTE and power switch to ON.
2. Press the TALK button on the Data Set.
3. Remove the handset and dial the computer telephone number.
4. When the computer responds with a high pitched tone, press the DATA button on the Data Set; replace the handset in its cradle.
5. Type return linefeed and log on.

Note: When connection is via telephone lines, the user must log on within a period determined by the system operator (nominally two minutes).

LINE/LOCAL SWITCH. Nearly all terminals will have a LINE/LOCAL or REMOTE/LOCAL switch. This switch should always be set to the LINE or REMOTE position.

FULL DUPLEX/HALF DUPLEX SWITCH. Some terminals have a FULL DUPLEX/HALF DUPLEX or ECHO/NO ECHO switch or jumper strapping. This should usually be set to the FULL DUPLEX or NO ECHO position. If this setting is improperly made, either the terminal will not print the characters that you type or the characters will be duplicated.

AUTO LF KEY. Some terminals have an automatic linefeed key. In all terminals except the HP 2644, this key must be off. On an HP 2644 terminal, this key should be on (depressed) in order to get the system's attention since that terminal does not have a *linefeed* key. If AUTO LF is not on, you may press the control-J keys whenever a linefeed is required; control-J generates the ASCII code for line feed.

TERMINAL SPEED. Terminals and modems vary in the speed with which they send and receive characters. The system will automatically detect the speed of your terminal and adjust its transmission speed accordingly. The system will accept terminal speeds of 110, 150, 300, 600, 1200, and 2400 bits per second (baud rate). When using a modem with a remote terminal you must use the same speed setting for both the terminal and the modem.

LOGGING ON AND OFF THE SYSTEM

Once connection has been made you must type *return linefeed*. On an HP 2644 that has no *linefeed* key, set AUTO LF on before typing *return* or press the control-J keys. This allows the system to determine the speed and parity of your terminal. The system should respond by typing the following message:

PLEASE LOG IN

Note that if you are using a non-ASCII character terminal such as the IBM 2741, you must not type *return linefeed* to get this message. You must, instead, press the attention key *ATTN*. On non-ASCII terminals that generate CALL/360 code, the PLEASE LOG IN message is garbled. The system detects the terminal code type when it receives the H in the HELLO command. The first character typed after PLEASE LOG IN must be the letter H.

HELLO COMMAND. The HELLO command is used to log onto the system. The command is followed by your account idcode, password, and terminal type parameter.

HELLO-H200,JOHN,1

H200 is the account idcode, JOHN is the account password, and 1 is the terminal type. Note that a comma is used to separate parameters.

Your *account idcode* is assigned to you by the system operator. It consists of a single letter followed by a three-digit number. As described above under Account and Library System, your account idcode controls library access.

Your *password* is linked to your account when your account is created by the system operator and can be modified at any time. The password consists of one to six characters; it may be kept confidential by the use of non-printing control characters. Such characters are entered by holding the control key down and pressing another character at the same time. Control characters are shown in this manual by a superscript. For example, control A is shown as A^c, control Z as Z^c, and so forth. The following characters may not be used as part of a password: S^c, X^c, H^c, J^c, M^c, and @^c (NUL or ASCII zero). Certain terminals associate special meanings with some characters. Such characters should not be used in passwords (refer to the HELLO command, Section X).

The *terminal type* parameter tells the system what type of terminal you are using. This information is used to modify the transmission of data to suit your terminal. Failure to specify the correct parameter may result in the loss of characters at your terminal or by the system. A list of terminal types with commonly used terminals associated with each type is provided below. A full discussion of terminal types is provided in Appendix D, Terminal Interface.

Terminal Type	Typical Terminals
0	HP2749A, ASR 33, ASR 38 teleprinters, HP3071A
1	HP2640A, HP2644A
2	HP2640A, HP2644A in page mode
3	HP2600A CRT terminal
4	HP2762A/B, GE TermiNet 300, GE TermiNet 1200
5	ASR 37
6	GE TermiNet 30
7	Texas Instruments Silent 700
8	Execuport 300
9	IBM 2741 Communication Terminal

A terminal type (0-8) entered with the HELLO command is associated with each of the nine ASCII code terminals. Type 9 is associated with non-ASCII generating terminals; this type need not be specified, but is returned if requested by the SYS function.

If no value is given for the terminal parameter the system assigns 0 as the default value. If you are unsure of the parameter value for your terminal, consult your system operator.

ERRORS DURING LOGGING ON. If you make a mistake when logging on, the system responds with an appropriate error message. For example, if you forget to type the hyphen while entering the HELLO command:

HELLOH200,JOHN,1

the system responds with the message:

ILLEGAL FORMAT

Re-enter the command in the correct form.

If the wrong password is entered:

HELLO-H200,JHN,1

the system responds:

ILLEGAL ACCESS

Re-enter the command with the correct password.

The messages **ILLEGAL ACCESS** and **ILLEGAL FORMAT** indicate that some or all of the current input is not acceptable to the system.

Spelling mistakes, format errors and incorrect parameters can be corrected while the line is being entered if the error is noticed before return is pressed. The control-H character (H^c) can be used to correct a few characters just typed, or the control-X character (X^c) can be used to cancel the entire line and start over.

Suppose the command **HELLO** is misspelled during entry. You may back up to the misspelled character by holding down the control key and typing H for each character you want to delete. On some terminals another backspace character is used (refer to Appendix D, Matrix of Terminal Types for particulars). Most CRT terminals backspace the cursor one character for each H^c entered; certain hardcopy terminals do not backspace but print an underline or back arrow for each character backspaced. In either case, the deleted characters are not entered in the line and you may then enter the correct characters to finish the line. To backspace four characters on a teletype and then enter the characters correctly:

HELO-H2 ← ← ← ← LO-H206,JOHN1,1 return

On most CRT terminals, the cursor is moved back to a position below the last character deleted.

HELO_ H2

You may then enter the correct characters starting at the cursor position below the "O". The correct characters replace the deleted characters:

HELLO-H200,JOHN1,1 return

ECHO COMMAND. The **ECHO** command can be used to adjust the data transmission to your terminal. If your terminal automatically prints characters that it sends, you can type:

ECHO-OFF return

to prevent the system from sending out duplicate characters. You can enter

ECHO-ON return

to cause the system to return data it receives to your terminal. This command is normally used when your terminal does not have an **ECHO/NO ECHO** or a **FULL DUPLEX/HALF DUPLEX** switch. **FULL DUPLEX** and **NO ECHO**, like the **ECHO-OFF** command, causes your terminal to suppress printing of the characters you enter. Normally, the system echoes entered characters at the terminal.

TIME COMMAND. The TIME command can be used to find out how much terminal time has been charged to your account.

TIME return
C100 ON PORT #05 FOR 00025 MIN. 00125 MIN USED OF 65000 PERMITTED.

MESSAGE COMMAND. The MESSAGE command allows you to enter a line of text at your terminal and have it sent to the system operator. For example:

MES-PLEASE ASSIGN A CARD READER TO C901 return

This message will appear on the system console along with a number identifying your port.

BYE COMMAND. When you have completed a session at your terminal, log off the system using the BYE command. For example:

BYE return

The system will respond by printing the total number of minutes that you were logged on.

0014 MINUTES OF TERMINAL TIME

You may instead log on at the same terminal using the same or another account. This will result in the first account being logged off.

YOU AND THE SYSTEM OPERATOR

Certain procedures that affect your use of the 2000 system can be performed only by the system operator at the system console. Table 1-2 contains a partial list of these operations. Depending on the operating policies at your site, you may ask your system operator to perform these operations. You can use the MESSAGE command to inform the operator of your request or your site may expect you to use special request forms.

You can ask the system operator, for instance, to assign additional capabilities to your account so that access to your library files and programs can be made available to users in other accounts.

A complete description of all the operations available to the system operator is contained in the HP 2000 System Operator's Manual (22687-9005).

Table 1-2. Selected System Operator Commands

COMMAND	THE SYSTEM OPERATOR USES THE COMMAND TO:
ANNOUNCE	Send a one-line message to one or all ports.
ASSIGN	Assign a non-sharable device to one user, to all users, or to the RJE facility; logically remove the device from the system.
BESTOW	Transfer programs and/or files from one user's library to that of another user.
BREAK	Allow a user to use the break key capability of his port even if it was programmatically disabled.
CHANGE	Modify the capabilities of an existing user account.
COPY	Produce a copy of a program or file from one user's library in that of another user's library.
KILLID	Remove a specified account from the system.
NEWID	Enter a new account on the system.
PHONES	Reset the time permitted users for logging on.
PURGE	Remove from all user's libraries the programs and files that have not been accessed since a given date.
RESET	Change the total terminal time recorded for a user or all users.

INTRODUCTION TO BASIC PROGRAMMING

SECTION

II

This section provides an overview of programming on the 2000 System. It discusses expressions used in BASIC statements, selected statements used in elementary BASIC programming, and how to use commands to manipulate programs in your work space or library. Specialized topics such as arrays, strings, files, formatted output, and system security are discussed as separate topics in other sections. Sections X and XI contain a rigorous definition of each of the 2000 BASIC commands and statements.

CONSTANTS, VARIABLES, AND EXPRESSIONS

The statements in 2000 BASIC deal with data in the form of decimal numbers, character strings, and logical values. Any of these types of data can be combined into expressions; numeric and logical values into numeric expressions and character strings into string expressions.

The data may be a specific value or *constant*. It may also be referred to by name as a *variable*. A constant has only one value whereas a variable may have any number of values, but only one at a time.

CONSTANTS

In accordance with the three types of data, there are three types of constants: numeric, string, and logical.

NUMERIC CONSTANTS. The numbers used in 2000 BASIC are entered or returned in any of three forms: integer, fixed-point, or floating-point. Within the computer, all numbers are represented as floating-point real numbers whose magnitude is between 10^{-38} and 10^{38} , zero, or between -10^{38} and -10^{-38} . Numbers that exceed these limits are converted to the nearest representable number; any number less than 10^{-38} is replaced by zero. When such conversion is needed, the system issues a warning message.

Besides magnitude, numbers are limited by precision, that is, the number of digits in an integer or fixed-point number or in the magnitude portion of a floating-point number. In 2000 BASIC, the precision of any number is six or seven decimal digits (23 binary bits). Numbers that exceed the system's precision are rounded to the nearest six or seven digits.

Integer Representation. The first or *integer* form consists of a positive or negative whole number:

```
12
+269
-10
32758
```


Commas are not allowed in an integer. This means that the last number cannot be expressed as:

32,758

Fixed-Point Representation. The second form is *fixed-point* consisting of a positive or negative number containing a decimal point:

1.2
.00269
-2.0
+25.689

Floating-Point Representation. The third form in which numbers are represented is *floating-point*. This representation allows you to express very large or very small numbers in a concise manner. In 2000 BASIC any number that cannot be represented by six decimal digits is represented as floating-point.

Using floating-point notation, any very large number can be represented as a smaller number multiplied by 10 raised to a power. For example 3 million (3000000) is expressed as 3 times 10^6 . Similarly, a very small number is expressed as a larger number multiplied by 10 raised to a negative power. For example, .000003 is expressed as 3 times 10^{-6} .

The 2000 BASIC floating point notation consists of an integer or fixed-point number followed by the letter E and an optionally signed integer. The number preceding E is a magnitude that is multiplied by some power of 10. The integer following E is the exponent, that is, the power of 10 by which the magnitude is multiplied.

To illustrate:

1E+23 = $1 \times 10^{23} = 100000000000000000000000$
1.0E23 (same as above)
.001E26 (same as above)
1.02E+4 = $1.02 \times 10^4 = 10200.$
1.02E-4 = .000102

STRING CONSTANTS. A string constant consists of a sequence of characters from the complete set of ASCII characters enclosed within quotes:

“ABC”
“**WHAT A DAY**”
“ X Y Z ” (spaces count, so this string is 7 characters)
“” (a null, empty, or zero-length string)
“ ” (a string consisting of two blanks)
“12.95” (a string containing numbers is not a number)

The last string although it contains a fixed point number cannot be used in arithmetic operations. It is simply a string of characters.

Any ASCII character may be represented in a string constant. Because a computer does not manipulate characters directly, string data is maintained internally as a sequence of numbers. Each character has its numeric equivalent (refer to Table A-1 in Appendix A for a list). By using the numeric equivalent of a character preceded by an apostrophe, even a quote can be specified in a string. Thus, according to Table A-1, "a" may be represented as '97, "A" as '65, and the quote itself is '34. One use of this format is to represent a character that is not otherwise representable, such as the control characters that have no printing graphic. The two types of string literals may be combined. For example:

```

"ABC"'68'69'70           is the string ABCDEF
"STOP"'13'10            is the string STOP return linefeed

```

LOGICAL CONSTANTS. The system uses two logical values, 1 for true and 0 for false. When tested, any non-zero value (positive or negative) is interpreted as true.

VARIABLES

A variable is a name used to reference a value. Unlike a constant, the value associated with a variable may change during program execution. A reference to a variable acts as a reference to its current value. Initially, the values of all numeric variables in a program are undefined and the program must assign a value to each one before it can be referenced. An error message is issued and the program terminated if the program attempts to use an undefined variable.

Variables may contain either numeric or string values.

Numeric variables are a single letter (from A to Z) or a letter followed by a digit (from 0 to 9):

```

A      A0
P      P5
X      X9

```

A variable of this type, when defined, contains a numeric value represented in the computer as a floating point number. Numeric variables can be interpreted as either false (zero value) or true (non-zero value).

String variables that contain string constants differ from numeric variables in that they consist of a letter (A to Z) followed by a dollar sign (\$) or of a letter followed by a digit (0 or 1) and a dollar sign:

```

A$     A0$
P$     P1$
X$     X0$

```

The length of any string variable must be defined in a DIM statement unless its maximum length is 1 character. (Refer to Section IV.)

SUBSCRIPTED VARIABLES. When a numeric variable references an array of numbers (refer to Section III), it may be subscripted to indicate a particular value within the array. If the array has one dimension, only one subscript is allowed. Elements in a two-dimensional array are identified by two subscripts separated by a comma. Note that array names may be only a single letter, not a letter and digit.

A(2) the second element of one-dimensional array "A"

B(2,4) the element corresponding to the 2nd row, 4th column of array "B"

The subscripts themselves can be represented as variables:

A(N) the Nth element of array A

or as expressions:

A(N,N/2) the element in the row N and column N/2 of array A

N/2 is an expression (see below) that when evaluated is rounded to an integer.

Strings may also be subscripted, but in this case, the subscript refers to a character position within the string starting at the left with position one:

A\$(2) start string in second character of A\$, continue to end of A\$

A\$(2,6) start string in second character of A\$ continue through 6th character

Section IV contains a full description of strings and how they may be referenced.

EXPRESSIONS

Expressions are fundamental elements of BASIC statements. A numeric expression combines constants, variables, and functions with operators in such a way that it can be evaluated to produce a single value. We have already discussed constants and variables. Another type of expression, called a function, is a named expression that, like a variable, has a value that may change during program execution. A detailed description of functions follows the discussion of expressions.

Numeric expressions used in BASIC are similar in form to normal algebraic expressions. Thus, most arithmetic operations can be programmed directly without changing their form. Even complex calculations can be programmed in a concise manner.

A string expression is a string variable, a string constant, or one of the two string functions, CHR\$ or UPS\$.

Some examples of expressions are:

$(P + 5)/27$ P is a variable that must have been previously assigned a value. 5 and 27 are constants. The slash is the divide operator. Parentheses group those portions of the expression evaluated first.

Suppose that the current value of P is 49, the expression $(49+5)/27$ is evaluated as the integer value 2.

$(N-(R+5))-T$ N, R, and T must all have been assigned values. + and - are the add and subtract operators. The innermost parentheses enclose the part evaluated first.

If the current value of N is 20, R is 10, and T is 5, the expression $(20-(10+5))-5$ results in the value zero.

OPERATORS. An operator performs a mathematical or logical operation on one or two values resulting in a single value. Generally, an operator is between two values, but there are unary operators that precede a single value. For instance, the minus sign in $A - B$ is a binary operator that results in subtraction of B from A; the minus sign in $-A$ is a unary operator indicating that A is to be negated.

Three types of operators are used on the 2000 BASIC system: arithmetic, relational, and logical:

Arithmetic Operators. All of the usual arithmetic operations are available in BASIC. The symbols "+" and "-" are used both as unary and binary operators. Unlike algebraic notation, implied multiplication does not exist in BASIC. Thus $A \times B$ must be written as $A*B$ rather than just AB. The operation of raising a number to a power also requires an explicit operator. Thus A^B is written as either $A**B$ or $A\uparrow B$ (the two forms are interchangeable). Two special operators, MIN and MAX, return the minimum or maximum respectively of their two operands. The table below lists the two unary and seven binary arithmetic operators recognized by BASIC.

Operator	Operation	Example
+	Unary plus (no effect)	+A
-	Unary minus (negation)	-A
*	Multiply	A*B
/	Divide	A/B
+	Add	A+B
-	Subtract	A-B
** or \uparrow	Exponentiate	A**B or $A\uparrow B$
MIN	Minimum	A MIN B
MAX	Maximum	A MAX B

Relational Operators. Relational operators are most often used to make decisions concerning the flow of control in a program (refer to the IF statement) but are also valid expression operators. They compare the values of two operands to see if they bear the relationship to each other signified by the operator. If the relation is true, the result value is

'true'; if not, the result value is 'false'. In most programs it is convenient to think of relational operators as producing logical values as their result. However, remember that the value 'true' is actually the number 1 and 'false' is actually the number 0. The table below lists the six binary relational operators.

Operator	Relation Tested	Example
=	Equality	A=B
< > or #	Inequality	A < > B
<	Less than	A < B
>	Greater than	A > B
<=	Less than or equal to	A < = B
>=	Greater than or equal to	A > = B

Logical Operators. Logical operators are useful in testing multiple relations or performing Boolean operations on 'true' and 'false' values. Like the relational operators, the result of a logical operator is 'true' or 'false' (1 or 0), depending upon the values of its operands. Remember that an operand is considered to be 'false' if its value is 0 or 'true' if its value is not 0. The table below describes the action of the logical operators.

Operator	Result	Example
NOT	'true' if operand is 'false'; 'false' if operand is 'true'	NOT A
AND	'true' if both operands are 'true'; 'false' if either operand is 'false'	A AND B
OR	'true' if either operand is 'true'; 'false' if both operands are 'false'	A OR B

EVALUATING EXPRESSIONS. An expression is evaluated by replacing each variable or function with its value and performing the operations indicated by the operators. The order in which operations are performed is determined by the operator order presented below.

**	↑		(highest)
Unary -	Unary +	NOT	
*	/		
+	-		
MIN	MAX		
Relational (=, <, >, <=, >=, <>, #)			
AND			
OR			(lowest)

The operator at the highest level is performed first followed by any other operators in the hierarchy. If operators are at the same level, the order is from left to right. Parentheses can be used to override this order. Operations enclosed in parentheses, which must be legal expressions, are performed before any operations outside the parentheses. When parentheses are nested, operations within the innermost pair are performed first.

For instance: $5 + 6*7$ is evaluated as $5 + (6 \times 7) = 47$

$7/14*2/5$ is evaluated as $((7/14) \times 2)/5 = .2$

In general, an expression alternates operators and operands. In addition each operand may be preceded by a unary plus or minus unless the preceding operator is ****** (or **↑**). In order to perform negative exponentiation, you may use parentheses as follows: $A**(-B)$.

In the following examples, let $A=1$, $B=2$, $C=3$, $D=3.14$, and $E=0$.

$A+B*C$ is evaluated as $A + (B \times C) = 7$

$A*B+C$ is evaluated as $(A \times B) + C = 5$

$A+B-C$ is evaluated as $(A+B) - C = 0$

$(A+B)*C$ is evaluated as $(A+B) \times C = 9$

$A \text{ MIN } B \text{ MAX } C \text{ MIN } D$ is evaluated as $((A \text{ MIN } B) \text{ MAX } C) \text{ MIN } D = C = 3$

In a relation, the relational operator determines whether the relation is equal to 1 (true) or 0 (false):

$(A*B) < (A-C/3)$ is evaluated as 0 (false) since $A*B=2$ which is not less than $A-C/3=0$.

Note that $5 < A < 10$ is always true. When evaluated from left to right $5 < A$ evaluates to either 0 or 1 depending on whether 5 is less than A; in either case, continuing the evaluation, 0 or 1 is less than 10.

In a logical expression, other operators are evaluated first for values of zero (false) or non-zero (true). The logical operators determine whether the entire expression is equal to 0 (false) or 1 (true):

$E \text{ AND } A-C/3$ is evaluated as 0 (false) since both terms in the expression are equal to zero (false).

$A+B \text{ AND } A*B$ is evaluated as 1 (true) since both terms in the expression are different from zero (true).

$A=B \text{ OR } C=\text{SIN}(D)$ is evaluated as 0 (false) since both expressions are false (0).

$A \text{ OR } E$ is evaluated as 1 (true) since one term of the expression (A) is not equal to zero.

$\text{NOT } E$ is evaluated as 1 (true) since $E=0$.

FUNCTIONS

A function names an operation that, when performed, produces a single value result. A numeric function is identified by a three-letter name followed by one or more parameters in parentheses. In most cases, only one parameter is used and this parameter is a numeric expression.

Since a function results in a single value, it can be used anywhere in an expression where a constant or variable can be used. To use a function, the function name followed by a parameter value in parentheses is placed in an expression. The function is evaluated and the result is used in the evaluation of the expression.

Examples of common functions:

SQR(x) where x is a numeric expression that results in a value ≥ 0 . When called, it returns the square root of x. For instance, if $N = 2$, $SQR(N+2) = 2$.

ABS(x) where x is any numeric expression. When called, it returns the absolute value of x. For instance, $ABS(-33) = 33$.

These functions are predefined. They are recognized automatically by the system and are available to any program. A complete list of the predefined functions for 2000 BASIC is provided in Section XI. In addition to the pre-defined functions, you may define and name your own numeric functions should you want to refer to a frequently repeated operation by name. You use the DEF statement (Section XI) in order to define a function. User-defined function names always begin with the letters FN followed by one of the letters of the alphabet from A through Z. Thus, you may define up to 26 functions in any program.

The following function definition defines a function (FNC) that finds the area of a circle from its radius.

```
10 DEF FNC(R)=3.1416*R**2
```

Once defined, your function can be used in an expression like any of the pre-defined functions.

All user-defined functions must result in a numeric value. Only one parameter may be used and it must be a numeric expression. Most pre-defined functions are also of this type, but several numeric functions use one or more character strings as parameters, and are primarily useful in string operations. In addition, two pre-defined functions return a string value. These functions are identified by a three-letter name followed by a \$. The functions used in string manipulation or that return a string value are described in Section IV as well as being rigorously defined in Section XI.

STATEMENTS

A BASIC program is made up of one or more statements that perform or define operations on data or add narrative comments to the program. Each statement is preceded by a statement number that determines the order in which statements are executed and also provides a means for one statement to reference another. The statement number is an integer between 1 and 9999. Program execution begins with the lowest numbered statement and proceeds in ascending order. Statements need not be entered in the order of execution since they are not executed at the time they are entered.

```

10 PRINT "THE VALUE OF A IS"
20 PRINT A
  5 LET A= 10
30 END

```

These three statements are listed and executed in the order 5, 10, 20, 30 regardless of the order in which they were entered.

Each statement is entered as a separate line ended with *return*. The system will either accept the line, responding with a *line feed* or reject the line with the message ERROR. A short description of an error can be obtained by entering any character other than *return* or a blank following the ERROR message. Since rejected statements are not entered into your program, correction consists of retyping a correct statement on a new line. If you detect an error in a statement before you have typed *return* you can use the control-H (H^c) or control-X (X^c) characters to alter or cancel the line as described in Section I. Statements already entered can be replaced by entering a new statement with the same statement number. Entering only a statement number followed by *return* deletes the numbered statement from the work space.

Statements are entered without strict format rules. Except where noted, the system ignores blanks within statements, commands, or data. This means that you can use blanks to make statements or input easy to read or leave them out entirely.

The following statement descriptions illustrate usage only; you must refer to Section XI for statement syntax.

ASSIGNMENT Statement

The assignment statement assigns a value to one or more variables. The value may be an expression, a constant, a function, or another variable of the same type. When the value is a string, then the variable to which it is assigned must also be a string; when the value is numeric, then each variable must be numeric.

For instance:

10	A=3	assigns the value 3 to the numeric variable A
20	A\$="3"	assigns the string value "3" to string variable A\$
30	B=A+1	evaluates expression A+1 and assigns result to B
40	C=SQR(B)	evaluates function SQR(B) and assigns result to C

The assignment statement is sometimes called a LET statement since the word LET is an optional part of this statement. For example, the statement 10 LET A=3 is identical in result to the statement 10 A=3.

When a value is assigned to more than one variable, the assignment is made from right to left. For instance, in the statement A=B=C=2, first C is assigned the value 2, then B is assigned the current value of C, and finally A is assigned the value of B. Since more than one variable can be assigned a single value, this statement provides a simple way to initialize variables at the start of a program:

```
50 LET X=Y7=Z=Z1=0
```

When a variable to be assigned a value is subscripted, the subscripts are evaluated first from left to right, then the expression is evaluated and the resulting value moved to the variable. For example:

```
70 LET N=2
80 LET A[N]=A[N+1]=9
```

*N is assigned the value 2.
Array elements A(3) and A(2) are assigned the value 9 in that order.*

Another example:

```
70 N=2
80 A[N]=5
90 A[A[N]]=A[N]=6
```

*A(2) is set equal 5.
A(2) and A(5) are set equal to 6 in that order.*

Refer to the LET statement in Section XI for the syntax and a further description of this statement.

PRINT Statement

The PRINT statement provides a simple and direct means for BASIC programs to display values on your terminal. The items to be printed are evaluated and displayed in sequential order from left to right. If there is more than one item to be printed, a comma or a semicolon separates each item. The choice of comma or semicolon affects the output format.

When a semicolon separates items, each item is printed immediately following the preceding item. When a comma separates items, each item is printed at the beginning of a field. The output line is divided into five consecutive fields: four of 15 characters and one of 12 characters, for a total of 72 characters. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line.

When a comma or semicolon follows the last item in a PRINT statement, the first item in the next PRINT statement is printed as a continuation of the current line. Otherwise, the next PRINT statement begins printing values at the start of a new line. A PRINT statement with no items to be printed either completes the previous line, if it ended with a comma or semicolon, or skips the current line. This is frequently used to produce a blank line.

PRINTING NUMERIC ITEMS. Each numeric item is followed by at least one blank so that there is always at least one space between items. Additional blanks are appended where necessary to make the total number of characters in a numeric item either 6, 9, 12, or 15. The number of characters depends on the magnitude of the item. (Refer to Numeric Output Formats in the PRINT Statement description, Section XI.) Leading zeros in a number are not printed. A positive number is preceded by a blank, a negative number by a minus sign. Trailing zeros after a decimal point are printed as blanks unless floating-point format is necessary, in which case, all six digits of the magnitude are printed.

To illustrate:



```
10 PRINT +4.0;-.025000000;-.0250001;999999.4;999999.6
20 PRINT +4.0,-.025000000,-.0250001,999999.4,999999.6
```

When executed, these statements produce the following print lines:

```
4  -.025          -2.50001E-02      999999.      1.00000E+06
4  -.025          -2.50001E-02      999999.      1.00000E+06
```

PRINTING STRING ITEMS. Each character within the quotes in a string constant, or the current value of a string variable may be printed with the PRINT statement. The separators, comma and semicolon, have the same effect with string items as with numeric items. However, a separator can be omitted when one or both of the items is a quoted string. A semicolon is assumed when the separator is omitted.

Since blanks are significant within strings, the number of characters printed is exactly the number in the string; no trailing or leading blanks are added to the printed item. As a result, if a semicolon separates consecutive strings, the second string is printed immediately following the first. If a comma follows a string, then the last of the fields it occupies is filled with blanks.

To illustrate:

```
10 LET A=-100
20 LET B0=0
30 LET B1=1000
40 PRINT "A="A,"B0 AND B1 ARE"B0"AND"B1
50 END
```

When executed, line 40 prints the following line:

```
A=-100          B0 AND B1 ARE 0      AND 1000
```

PRINT FUNCTIONS. Up to this point, the comma separator provides the only control over the format of data displayed by a PRINT statement. 2000 BASIC provides a method for requesting completely precise formatting (refer to the PRINT USING statement in Section VI) and also provides simple control facilities with three predefined functions. The functions TAB, SPA, and LIN may be specified as print items. These functions are called print functions since they may appear only in a PRINT statement.

The TAB function spaces to the print position specified as the TAB argument. For example:

```
10 PRINT TAB(20)"ACCOUNTS PAYABLE"
```

spaces to print position 21 and then prints the string constant.

The 72 print positions in a line are numbered from 0 to 71. If the current line is already at or past the specified position, no action is taken. If the specified position is greater than 71, the print position is moved to the beginning of the next line.

The SPA function prints the number of spaces specified as the SPA argument. For example, 10 PRINT A,SPA(20),B puts 20 spaces between the end of A and the beginning of B. If the number of spaces will not fit on the current line or is greater than 71, the current line is completed followed by a carriage return and line feed.

The LIN function specifies the number of line feeds to generate. If the argument is positive, a carriage return is generated as well as the specified number of line feeds. If negative, LIN advances the indicated number of lines but preserves the current character position. For example:

```
10 PRINT "ABC"LIN(-1)"DEF"LIN(2)"GHI"
```

results in the following lines:

```
ABC
  DEF
GHI
```

A typical use of all three functions is to provide header information for a report:

```
20 PRINT TAB(10)"SUMMARY REPORT"SPA(15)"PAGE 1"
30 PRINT LIN(3)"DETAIL LINES"
```

When executed, these lines are printed as:

```
                SUMMARY REPORT                PAGE 1

DETAIL LINES
```

LIN(0) generates a carriage return without a linefeed. This allows overprinting of a previous entry.

REM Statement

The REM statement allows you to insert explanatory comments within your program. The statement is not executed and does not affect program execution. Like any other statement, REM is preceded by a statement number. All characters, including blanks, that appear after the word REM are the remark.

The remarks introduced by REM are saved as part of the BASIC program and are printed when the program is listed or punched. They are, however, ignored when the program is executed. If control is passed to the REM statement, it continues to the following statement with no other effect.

Remarks are easier to read if REM is followed by spaces or punctuation marks as shown in the following examples:

```

10  REM: THIS IS AN EXAMPLE
20  REM  OF REM STATEMENTS
30  REM -- ANY CHARACTERS MAY FOLLOW "REM": "//+!&, ETC.
40  REM .. REM STATEMENTS ARE NOT EXECUTED.

```

GOTO Statement

A program is executed normally in the order determined by statement numbers; the lowest is executed first, the highest last. The GOTO statement overrides this normal sequential order by transferring control to a specified statement. The statement to which control transfers must be in the program.

If the GOTO transfers to a statement that cannot be executed (such as REM), control passes to the next sequential statement.

The GOTO statement can specify a single statement number or it can specify a list of statement numbers to which control branches depending on the value of an expression. For instance, in the statement 10 GOTO X OF 20,30,60 control branches to statement 20 if the value of X is 1, to statement 30 if the value of X is 2, and to statement 60 if the value of X is 3. The value of X is rounded so that it evaluates exactly to an integer. If this integer value is less than 1 or greater than the number of statement numbers in the list, then the statement is ignored and the next statement executed.

The following example shows a simple GOTO statement in line 200 and a multi-branch GOTO in line 600:

```

100  LET I=0
200  GOTO 600
300  REM .. PRINT INITIAL VALUE
400  PRINT "THE INITIAL VALUE OF I IS" I
500  LET I=I+1
600  GOTO I+1 OF 300,500,500,800
700  REM .. PRINT FINAL VALUE
800  PRINT "THE FINAL VALUE OF I IS" I
900  END

```

When executed, this program prints:

```
THE INITIAL VALUE OF I IS 0  
THE FINAL VALUE OF I IS 3
```

END/STOP Statements

The last executed statement in every program must be an END statement. A program that is not terminated by an END statement will not execute. In addition to being the program terminator, an END statement is also used to stop program execution wherever it appears, at the end of the program or within the program.

The STOP statement, like the END statement stops program execution. However, STOP is specified within a program to halt program execution before the last statement; it is never the last statement.

Program execution can be halted before the end with a GOTO statement that transfers control to the last or END statement. Thus there are several ways to halt program execution, but only the END statement may be the last statement in the program.

To illustrate, the following three sequences all have the same effect:

10	10	10
⋮	⋮	⋮
500 STOP	500 GOTO 900	500 END
⋮	⋮	⋮
900 END	900 END	900 END

IF Statement

The IF statement provides another means of altering the sequence of execution in a program. An expression following the word IF is evaluated and if it is logically true (non-zero), then control passes to a specified statement number. If the expression is logically false (zero), control falls through to the next sequential statement after IF.

For example, when the following statement is executed:

```
10 IF A >= 100 THEN 50
```

the value of A is tested and if it is equal to or greater than 100, statement 50 is executed. If it is less than 100, the next sequential statement is executed. The statement number following THEN must be a valid statement in the program. It need not be executable; if it is not, then control passes to the next executable statement.

The following short program computes the squares of the numbers 1 to 10 and stores them in array A:

```

10 LET N=1
20 A[N]=N*N
30 PRINT A[N];
40 LET N=N+1
50 IF N <= 10 THEN 20
60 END

```

When executed, the squares are printed as follows:

```

1      4      9      16     25     36     49     64     81     100

```

This example illustrates use of IF to control a program loop by setting up a condition that completes the loop. IF is also a powerful tool for choosing alternate logic paths or making any decision needed to accomplish a program's purpose.

A relational operator is not needed in an expression that is tested for zero or non-zero. For example the following pairs of statements are exactly equivalent:

```

10 IF A<>0 THEN 50
10 IF A THEN 50           Go to statement 50 if A not zero.

10 IF A=0 THEN 50
10 IF NOT A THEN 50      Go to statement 50 if A is zero.

```

FOR/NEXT Statements

A pair of statements, FOR and NEXT, allow the repetition of a group of statements. The FOR statement precedes the statements to be repeated, and the NEXT statement directly follows them. The number of times the statements are to be repeated is determined by the value of a simple numeric variable specified in the FOR statement. Although the IF and GOTO statements can usually perform the same type of looping as FOR and NEXT statements, FOR and NEXT are generally simpler to follow.

The following program illustrates a FOR/NEXT loop; note that it performs the same function as the program used to illustrate the IF statement:

```

10 FOR N=1 TO 10
20 A(N)=N*N
30 PRINT A(N);
40 NEXT N
50 END

```

The variable N in the FOR statement is called the control variable. When FOR is executed the first time, N is set to 1 (the initial value) and is then tested to determine if it is greater than 10 (the final value). If N is greater than 10 control passes to the statement after NEXT, otherwise, the statements between FOR and NEXT are executed. When NEXT is executed, the control variable (N) is incremented by 1 and control returns to the FOR statement. N is tested again and the following statements are repeated until N is greater than 10.

In this form of the FOR statement, the control variable is always incremented by 1. A step value can be included in the statement that will cause the control variable to be incremented by that value or, if the value is negative, to be decremented. For example:

```
10  FOR J=10 TO 1 STEP -1
20  PRINT J;
30  NEXT J
40  END
```

J is initially set to the value 10 and following each execution of statement 30 it is decremented by 1 until it is less than 1. When executed, the program prints:

```
10   9   8   7   6   5   4   3   2   1
```

To summarize, the following steps occur whenever a FOR/NEXT loop is executed:

1. The value of the FOR variable is compared to the final value; if it exceeds the final value (or is less when the STEP value is negative), control skips to the statement following NEXT.
2. Statements between the FOR statement and the NEXT statement are executed in normal program order.
3. At NEXT, the FOR variable is incremented by 1 or by the STEP value, if specified.
4. Return to step 1.

Pairs of FOR/NEXT statements can be used within other pairs of FOR/NEXT statements to produce loops nested within loops. Each nested loop must be completely contained within the outer loop. Also, the control variable of a nested loop must be different than the control variable of an outer loop.

The following program illustrates nested FOR/NEXT loops:

```
100  FOR I=1 TO 5
200  FOR J=1 TO 1 STEP -1
300  PRINT J;
400  NEXT J
500  PRINT
600  NEXT I
700  END
```

The number of times the inner loop is executed is determined by the value of I; each time I is incremented, the inner loop (J) is executed one more time before falling through:

```

1
2   1
3   2   1
4   3   2   1
5   4   3   2   1

```

A FOR/NEXT loop terminates normally when the value of the control variable exceeds the final value or is less than the final value if the STEP value is negative. A loop may also be terminated when control is transferred out of the loop with an IF statement, GOTO statement, etc. This causes no difficulty and the program can use the loop again by transferring control to the FOR statement. It is very bad practice, however, to transfer control to any statement after the FOR statement that is within the loop up to and including the NEXT statement. If this is done, unpredictable results can occur when control reaches the NEXT statement.

READ/DATA/RESTORE Statements

There are several other methods besides an assignment statement for assigning values to variables. The READ, DATA, and RESTORE statements allow you to set a large number of variables to different constant values. This can be done with assignment statements only if you write one statement for each value. The READ statement assigns constants specified in DATA statements to one or more variables. RESTORE allows the same data to be read again.

DATA statements each contain a list of constants separated by commas. One or more DATA statements may be specified. All the constants in the combined DATA statements constitute a data list. The list starts with the DATA statement having the lowest statement number and continues to the statement with the highest number. DATA statements can be anywhere in the program; they need not precede the READ statement, nor need they be consecutive.

A READ statement contains a list of simple or subscripted variables separated by commas. When a READ statement is executed, each variable is assigned a value from the data list. A pointer is kept within the program to indicate which constant is next in the list. If a variable in the READ statement list is numeric, the next item in the data list must be numeric; if a variable is a string, the next item must be a string. It is possible to determine the type of the next item with the TYP function (refer to Section XI).

At the beginning of program execution, the pointer to the data list is set to the first constant in the first DATA statement. As constants are assigned, the pointer is advanced consecutively through the data list.

The following example illustrates how READ and DATA operate:

```

10 DATA 3, 5, 7
20 READ A, B, C
30 PRINT A; B; C
40 END

```


When executed, the program prints the values of variables A, B, and C:

3 5 7

The RESTORE statement can be used to access constants in a non-serial manner by specifying a particular DATA statement to which the pointer is moved. When RESTORE is followed by a statement number, the pointer is moved to the first constant in the specified statement. If the statement is not a DATA statement, the pointer is moved to the first DATA statement with a higher statement number. When no label is specified, RESTORE restores the pointer to the first constant in the first DATA statement as illustrated in the following example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 PRINT A;B;C;D;E;F
50 DATA 5,10,15
60 END
```

When the program is executed, the values are printed:

5 10 15 5 10 15

When followed by a statement number, RESTORE specifies a particular DATA statement:

```
10 RESTORE 50
20 READ X,Y,Z
30 PRINT X;Y;Z
40 DATA 2.5,30.1,15
50 DATA 100,150,300,500,750
60 END
```

Variables X, Y, and Z are assigned the first three values in statement 50. When the program is executed, these values are printed:

100 150 300

An error occurs and the program terminates if a READ statement requests more values than remain in the list of DATA statements. The TYP function can be used to determine the end of the data list as well as the type of the next item.

When the TYP function is used to test a DATA list, you use an argument value of zero. The value of TYP(0) is returned as the value:

- 1 if the next item in the DATA list is numeric.
- 2 if the next item is a string value.
- 3 if there is no more data in the list.

Programs can take advantage of the ease with which BASIC statements can be changed. A general program can be supplied with new data for each execution by entering new DATA statements with the same numbers used before. In the following program, DATA statements can be supplied between statement numbers 500 and 600; the program performs calculations on whatever data is provided:

```

10 N=S=0
15 REM...TEST FOR END OF DATA
20 IF TYP(0)=3 THEN 70
30 READ T
40 N=N+1
50 S=S+T
60 GOTO 20
70 PRINT "AVERAGE IS" S/N
80 IF N >= 3 THEN 100
90 STOP
100 RESTORE
110 READ A,B,C
120 PRINT "AVERAGE OF FIRST 3 ITEMS IS"(A+B+C)/3
500 DATA 5.5,3.46,52
510 DATA 77.3,.89,50
600 END

```

When executed with the DATA statements shown above, the averages are calculated as:

```

AVERAGE IS 31.525
AVERAGE OF FIRST 3 ITEMS IS 20.32

```

Suppose new data is supplied as follows:

```

500 DATA 17.6,.009,305.67,28.95
510 DATA 59.75,175.6,33.59,72.0
520 DATA 37.98,20.75

```

When executed, the following averages are calculated:

```

AVERAGE IS 75.1899
AVERAGE OF FIRST 3 ITEMS IS 107.76

```

INPUT Statement

The INPUT statement allows you to enter data directly to an executing program from your terminal. An INPUT statement contains a list of variables separated by commas into which the data is entered.

When executed, INPUT prints a question mark (?) at the terminal to indicate it is ready for data. The program pauses and you may then enter one or more constants separated by commas. Numeric constants should be entered into numeric variables and string constants into string variables. The constants are assigned to the INPUT list from left to right. If you enter fewer constants than there are variables in the INPUT list, a double question mark (??) is printed to indicate more data is expected.

It is often a good idea to precede the INPUT statement with a PRINT statement that displays a message indicating the type of data expected. To illustrate:

```
10 PRINT "ENTER TWO NUMBERS";
20 INPUT A,B
30 PRINT "THEIR SUM IS";A+B
40 END
```

When executed:

```
ENTER TWO NUMBERS? 2.5, 5
THEIR SUM IS 7.5
```

If too many constants are entered, the extra constants are discarded and a message is sent to the terminal. Suppose you execute the same program but enter more than two constants:

```
ENTER TWO NUMBERS? 2, 5, 5
EXTRA INPUT - WARNING ONLY

THEIR SUM IS 7
```

Rather than respond to a program waiting for input, you can terminate its execution by pressing the BREAK key.

ENTER Statement

The ENTER statement is similar to the INPUT statement but provides more control over the input operation. You may request the port number of the terminal at which the response is entered, set a limit to the time allowed to respond and retrieve the actual response time. Only one data item is accepted by ENTER, unlike INPUT that allows you to enter a string of data items. Two other differences are that no prompt is printed to signal that data is expected, and no line feed is generated after the data is entered.

ENTER may be used simply to retrieve a terminal port number:

```
10 ENTER #N
20 IF N>10 THEN 40
30 PRINT "TERMINAL NUMBER IS"N
40 END
```

A terminal number can be between 0 and 31; this program prints the number if it is less than or equal to 10, for instance:

```
TERMINAL NUMBER IS 2
```

This terminal number feature may be used separately (as illustrated above) or in combination with data entry, or you may omit terminal number retrieval.

A common use of ENTER is to test students:

```
10 PRINT "WHAT IS .25 TIMES 75?"
20 ENTER 30,T,X
25 IF T <= -257 THEN 92
30 IF T=-256 THEN 100
40 IF X=.25*75 THEN 80
50 PRINT LIN(1)"SORRY, THE CORRECT ANSWER IS".25*75
60 PRINT "TRY THE NEXT PROBLEM"
70 GOTO 110
80 PRINT LIN(1)"CORRECT, YOU ANSWERED IN"T"SECONDS"
90 GOTO 110
92 PRINT "TRANSMISSION OR PARITY ERROR"
96 GOTO 110
100 PRINT "SORRY, TIMES UP, TRY THE NEXT PROBLEM"
110 REM .. THE NEXT PROBLEM COULD START HERE ...
120 END
```

When executed:

```
WHAT IS .25 TIMES 75?
18.75
CORRECT, YOU ANSWERED IN 4 SECONDS
```

In this example, the number 30 immediately following ENTER in statement 20 is the allowed response time in seconds. Any expression could be used that, when evaluated and rounded, results in an integer between 1 and 255. The variable, T, is the actual time taken for the response. If the time limit is exceeded, this variable is set to the value -256 and input is not accepted. If a parity error occurs during transmission, the variable is set to -257, if a character is lost, it is set to -258; in either case input is not accepted. The last variable in the list, X in the example, is assigned the single value requested by ENTER.

Note that the first PRINT statements following input start with the LIN(1) function to generate a linefeed and carriage return. This is done because the ENTER statement does not provide a line feed following input. If any output is to be printed after the input line, you must provide the line feed so that subsequent output does not overwrite the input line. You may, of course, choose to overwrite the input line for purposes of security.

The following example combines retrieval of the terminal number with timed input:

```
10 LET B=20
20 DIM A$(30)
30 PRINT "YOU HAVE "B"SECONDS TO ENTER YOUR NAME"
40 ENTER #A,B,C,A$
45 IF C=-256 THEN 60
50 PRINT LIN(1);A$ " IS USING TERMINAL #"A
55 GOTO 70
60 PRINT "TIME EXPIRED"
70 END
```

When executed:

```
YOU HAVE 20 SECONDS TO ENTER YOUR NAME
JOAN
JOAN IS USING TERMINAL # 2
```

GOSUB/RETURN Statements

When the same group of statements is used at several points in your program, you may write such statements once and transfer to them from different locations. Such statement groups are called subroutines. The GOSUB statement is used to transfer control to the first statement of such a subroutine; the RETURN statement returns control to the statement following GOSUB.

Like the GOTO statement described earlier, GOSUB can transfer to a single statement number or it can transfer to one of a list of statement numbers depending on the value of an expression.

Since BASIC subroutines are simply a collection of statements, it is a good idea to place them where they will not be executed in the normal sequence of program execution and to clearly indicate their purpose and beginning statement number with a remark. If RETURN is not the last statement, a remark should also indicate the end of the subroutine.

The following program portions use a subroutine to ask for the time of day:

```

100 L=23
110 PRINT "ENTER HOUR OF DAY"
120 GOSUB 1000
130 H=R
140 REM .. RESET LIMIT VARIABLE "L"
150 L=59
160 PRINT "ENTER MINUTE OF HOUR"
170 GOSUB 1000
180 M=R
190 PRINT "ENTER SECOND"
200 GOSUB 1000
210 S=R
.
.
.

1000 REM...BEGIN SUBROUTINE
1010 INPUT R
1020 IF R<0 OR R>L THEN 1040
1030 RETURN
1040 PRINT "IMPOSSIBLE VALUE, ENTER CORRECT VALUE"
1050 GOTO 1010
1060 REM...END SUBROUTINE
1070 END

```

Another subroutine can be called within a subroutine; that is, subroutines can be nested. When a RETURN is executed, it transfers to the statement following the last executed GOSUB. Up to twenty GOSUB statements can be executed without an intervening RETURN. Care should be taken when nesting subroutines since return to the statement following the first GOSUB will not occur until all intervening RETURN statements have been executed. You may of course use GOTO or IF statements to transfer directly into or out of a subroutine at any point.

The following example illustrates a multi-branch GOSUB statement in line 1020; the third subroutine to which it transfers contains a nested subroutine. Note that GOSUB may follow the subroutine to which it transfers:

```

.
.
.
90  GOTO 1010
100 REM START OF FIRST SUBROUTINE
110 LET X=SQR(A+25)
120 PRINT "X = "X
130 RETURN
140 REM END OF FIRST SUBROUTINE
150 REM START OF SECOND SUBROUTINE
160 LET Y=COS(X)
170 PRINT "Y = COSINE X = "Y
180 RETURN
190 REM END OF SECOND SUBROUTINE
200 REM START OF THIRD SUBROUTINE
210 LET Z=Y+X
220 PRINT "Z = X+Y = "Z
230 REM TRANSFER TO NESTED SUBROUTINE:
240 GOSUB 260
250 REM UPON RETURN HERE, RETURN TO STATEMENT 1030:
255 RETURN
260 REM FIRST STATEMENT OF NESTED SUBROUTINE
270 B=SIN(Z)
280 PRINT "SINE Z = "B
285 REM RETURN TO STATEMENT 250:
290 RETURN
300 REM END OF NESTED SUBROUTINE
.
.
.
1010 A=0
1020 GOSUB A+1 OF 100,150,200
1030 LET A=A+1
1040 IF A<3 THEN 1020
1050 END

```

When executed, the four subroutines print the following:

```

X = 5
Y = COSINE X = .283662
Z = X+Y = 5.28366
SINE Z = -.841213

```

DEF Statement

You may define your own functions with the DEF statement. Once defined in a program, the function can be referenced exactly as if it were one of the system pre-defined functions. The function definition must be in the same program that references the function; it may appear anywhere in the program, before or after the reference. Like other non-executable statements, control passed to a DEF statement will continue to the next sequential statement with no other effect.

A function name consists of the letters FN followed by one of the 26 letters of the alphabet. This permits you to define 26 functions in any program. The name is followed by a single argument within parentheses that may be any simple numeric variable. It may not be subscripted or be a string variable. The argument is followed by an equals sign which is followed by the expression that defines what the function does.

Any legal numeric expression can be used. The expression can contain references to another function as long as that function is fully defined elsewhere.

The following function definition defines FNT as the tangent function:

```
10 DEF FNT(X)=SIN(X)/COS(X)
```

When this function is referenced, an actual argument is used in the reference. This argument may be any numeric expression; it replaces the function argument, in this example the variable X, when the function is evaluated. It could be a simple variable:

```
20 INPUT B
30 PRINT "THE TANGENT OF"B"IS "FNT(B)
```

Or the argument could be an expression:

```
20 LET B=6
30 LET X=FNT(B*1.2)
```

The following program calculates the area of a circle from any radius using the function FNC(R). Note that the actual argument is a variable with the same name as the function variable.

```
10 PRINT "ENTER A RADIUS"
20 INPUT R
30 PRINT "AREA OF CIRCLE WITH RADIUS"R"IS"FNC(R)
40 DEF FNC(R)=3.1416*R ** 2
50 END
```

When executed, you may enter any numeric value for R:

```
ENTER A RADIUS
?350
AREA OF CIRCLE WITH RADIUS 350 IS 384846.
```


COMMANDS

You use commands to instruct the system to perform certain control functions. Commands differ from BASIC statements in both purpose and form. A command instructs the system to perform some action immediately, while a statement is an instruction to perform a particular function only when the program of which it is a part is executed. A statement is always preceded by a statement number, a command never is. Commands can be entered at any time except when the current program is executing. They are either accepted and executed or else rejected with an appropriate message.

Each command is a single word which can be abbreviated to its first three letters. Some commands also have optional or required parameters following them. If parameters are used, they are separated from the command name by a hyphen (-). Multiple parameters are separated from each other by commas. You signal completion of a command's entry by pressing *return*. If the command is misspelled or otherwise unrecognized, the system responds with three question marks (???).

Many commands do not produce a response, their completion is announced with a line feed. Others display one or more lines of information. In these cases you can stop command operation with the BREAK key. The message STOP indicates that the terminal is again available for entering statements or commands.

Section X contains a full description of all the commands provided with 2000 BASIC. This section describes those commands you use to manipulate programs in your work space, to save and retrieve programs in your library, and to prepare and use paper tape copies of programs.

YOUR WORK SPACE

Each terminal logged on the 2000 system is given a work space that is approximately 10,000 words long. (A word is a unit of length, two characters or 16 bits long.) Whenever you log on the system, this space is empty. During program preparation, you enter the statements of a program into this work space. When the work space contains an executable program, it can be run from the work space. The content of the work space is cleared when you log off from the system.

Since programs are entered one statement at a time, the statements in the work space do not always make up a complete program. The content of this space is referred to as the current program whether or not it constitutes an executable program.

USING YOUR WORK SPACE. You can control your work space in a variety of ways. You can change, display, save, delete, or give a name to whatever is held in this space. To illustrate these capabilities, suppose you enter the following statements:

```
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

This is a complete program. You may run the program as soon as it is entered, you may modify the program, or else list it with the LIST command.

If you want to change line 20, you simply re-enter the line:

```
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
```

If you list the program, the new line 20 replaces the previous line 20:

```
LIS
```

```
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```



Note that when you enter program statements, spaces are not significant. You can omit spaces between elements of a statement or put in more than one space. When the program is listed, 2000 uses a standard spacing between elements. Also, if you have entered statements out of order, 2000 orders them in ascending sequence by statement number and lists them in this order.

If you want to add statements, you can insert them by using statement numbers between those already used. For example, to insert a statement between lines 50 and 60:

```
55 IF A>0 THEN 70
```

At least one statement must now be added at line 70:

```
70 LET X=A/2
71 PRINT "A DIVIDED BY 2 IS"X
72 END
```

Another command, RENUMBER, allows you to renumber the statements in your work space. When no parameters are used, this command renumbers each statement starting with statement 10 in ascending order by 10. It also adjusts any statement references to the new numbering scheme. When you renumber and list the expanded program, note that the line entered as 55 is now line 60 and references line 80:

```
RENUMBER
LIS
```

```
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 IF A>0 THEN 80
70 END
80 LET X=A/2
90 PRINT "A DIVIDED BY 2 IS"X
100 END
```

You may renumber lines starting at a statement other than 10 and using any increment you please. For instance, to start at statement number 1 and increment by 5, you enter:

```
REN-1,5  
LIS
```

```
1  REM...FIND THE PRODUCT  
6  PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"  
11 INPUT B,C  
16 LET A=B*C  
21 PRINT A  
26 IF A>0 THEN 36  
31 END  
36 LET X=A/2  
41 PRINT "A DIVIDED BY 2 IS"X  
46 END
```

You can also choose to renumber only a portion of your program. Refer to Section X for the additional parameters that provide this feature.

You may delete any line of your program simply by typing the statement number of the line. If you want to delete groups of lines, it is easier to use the DELETE command than to type each line number. For example, to delete statements 26 through 41, you enter the command:

```
DEL-26,41
```

If the last statement number is the end of the program, you only need to specify the statement number at which deletions are to start.

If you now renumber your program and list it, you are back to the original statements you entered.

```
REN  
LIS
```

```
10  REM...FIND THE PRODUCT  
20  PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"  
30  INPUT B,C  
40  LET A=B*C  
50  PRINT A  
60  END
```

So far, the LIST command has been used to list the entire contents of your work space. It can also be used to list selected portions of the current program by specifying the first and last statement numbers to be listed as parameters to LIST. If the two numbers are the same, only one line is listed; if the second parameter is omitted, the list starts at the specified statement number and continues through the last statement in the work space. For example, to list line 30:

```
LIS-30,30
```

```
30 INPUT B,C
```

At any point while you are preparing a program in your work space, you can run it to check for problems. To execute the content of the work space, you use the RUN command. Execution starts with the first executable statement in the work space:

```
RUN
```

```
ENTER TWO NUMBERS SEPARATED BY A COMMA
?375,4
 1500
```

```
DONE
```

An executed program remains in the work space unchanged by execution and it can be run as often as desired until you log off and it is lost. However, you will usually want to save a completed program that executes correctly. Saved programs are kept in your library. But before a program is saved, you must name it so that you can subsequently recover it by name.

Use the NAME command to give your work space a name:

```
NAME-MYPROG
```

Now the program can be saved with the SAVE command:

```
SAVE
```

Note that you do not use the program name to save the program. The SAVE command saves whatever is in your work space as MYPROG. The NAME command actually gave the name MPROG to the used portion of your work space. For this reason, it is important before using the RUN, NAME, or SAVE commands to be sure that your work space contains only the program you want to execute, name, or save.

If there is not enough space in your library for the program, the program is not saved and a message informs you that your library is full. You may use the LENGTH command (Section X) to determine the length of your program and how much space remains in your library. It may be possible to shorten your program or to purge unused programs from your library.

The work space is always clear when you log on, but you can also clear it with the SCRATCH command. After you save MYPROG you might want to start developing another program. If you simply enter new statements, they will replace any statements in MYPROG with the same statement numbers, or else the new statements will be interspersed with MYPROG or be added to the end of it. So, it is a good idea once a program is saved to clear the work space with the SCRATCH command:

SCR

Two other commands, GET and EXECUTE, that bring a program from the library into the work space also clear this area.

YOUR LIBRARY

Each account has a private library. This library is used to store your programs and data files. The size of the library is set by the system operator. You may be able to make programs and files in your library accessible to other accounts and in addition you can access your group and system libraries. (A detailed discussion of libraries and library access is contained in Section VIII.)

ACCESSING YOUR LIBRARY. When you save a program with the command SAVE, it is stored in your account library. You may then retrieve this program with the GET command:

GET-MYPROG

GET first performs an implicit SCRATCH command, clearing the work space and its name, and then loads a copy of program MYPROG into the work area setting the name to correspond. Your library is not altered by GET. You may now modify, list, or execute the contents of your work area exactly as if you had entered the program a statement at a time.

You may also bring a program from your library into the work area without changing the current contents of this area. The APPEND command appends a copy of any available library program to the current program in your work space. Since library entries are not restricted to complete programs, you may find this command useful to incorporate previously written routines in a program being developed in your work space. The current program name is not changed by APPEND. The first statement of the library program must be greater than the last statement of the program under development. The RENUMBER command can often be used to lower the last statement number of your current program. It is also a good idea to use large numbers for any routines you expect to append to a subsequently developed program. Refer to Section X for the APPEND syntax and restrictions.

If you want to execute a program in your library, you can use the EXECUTE command. EXECUTE acts as a combination of the GET, RUN, and SCRATCH commands. That is, it clears your work space, brings the named program into the work space, executes it, and following execution, clears the work space again. For example:

```
EXE-MYPROG
MYPROG
```

```
ENTER TWO NUMBERS SEPARATED BY A COMMA
? 59, 12
  708
```

```
DONE
```

Note that programs in libraries other than your own can be retrieved and executed with the GET and EXECUTE commands. Any non-private program in your group library may be retrieved or executed by preceding the program name with an asterisk (*); any non-private program in the system library may be retrieved or executed by preceding the program name with a dollar sign (\$). Programs can be accessed by GET only if they are *unrestricted* or *protected*. *Locked*, as well as *protected* and *unrestricted* programs can be run with EXECUTE. Refer to Section VIII for a description of library access, in particular, user-imposed restrictions on programs.

When you no longer have any use for a particular program, you can remove it from your library with the PURGE command. The work space is not affected. A purged program can no longer be recovered unless a copy exists in another library or in some other form such as paper tape. The PURGE command is frequently used to delete another version of a program so that a new program can be saved under the same name.

Suppose you have no further need of MYPROG:

```
PUR-MYPROG
```

MYPROG can no longer be accessed and the space it occupied is returned to your account library.

LIBRARY CATALOG. A complete list is maintained by 2000 of all the programs and files in your library as well as those in the group and system libraries. The CATALOG command requests a list of the programs and files in your account library; the GROUP command of those in your group library; the LIBRARY command of those in the system library. Only those programs and files that you may access are listed.

The lists are in alphabetic order by program or file name and you may request that a list start with a particular sequence of letters or numbers (numbers precede letters). For example, CAT-P generates a list of all programs or files in your account library that begin with the letters P through Z. You may terminate the list at any point by pressing the BREAK key.

Introduction to Basic Programming

In addition to the program or file name, its length in records, and for a file, the number of words per record, the list uses code letters to indicate the type of file or program and any access restrictions. For example, FU following a file name indicates it is a BASIC formatted file with unrestricted access. Private programs saved with the SAVE command have a blank code or else a letter indicating an access restriction other than private. Refer to CATALOG command in Section X for the possible codes and their meaning.

The following examples show how to get a list of the system library files and programs accessible to your account, the group library for that same account, and a list of your account library starting with a particular name.

```
LIB
NAME      LENGTH RECORD  NAME      LENGTH RECORD  NAME      LENGTH RECORD
000015 FL      1      1         AL LP1      66  ASC25    AL LP0      66
AXXX     AL LP0      66  BARKPI   CU       9      BARKSI   CU       14
BASICS   P        2      CIMS     CL       1      COMPAR   CL       6
CONFG    AL 10      63  CREATE   U       19      DATTIM   U       3
DIREC    CL 11      DHP      CL       1      ED0000   FU      200
EDERR0   FU 50      EDIT00   CP      25      EDIT01   CP      29
EDIT32   CP 14      EDITOR   CP      28      F        AL       1      63
FAITH    FL 17      FCOPY    CP      27      FCOPY1   CP      26
FCOPY2   CP 28      FCOPYC   FP      14      FCOPYM   FP      50
FCR0     AL CR0      80  FILE     FP       1      FILE1    FL       1
FILSCN   CP 2      FINDI    U       7      FINDIT   U       5
FINDOR   U 6      FIXPTR   U       2      FJL0     AP JL0      67
FJT0     AL JT0      66  FLP0     AP LP0      66  FMT0     AP MT0      256
FPR0     AP PR0      64  FPR0     AP PR0      64  GDUMP    U        2
GEORGE   FL 100     HISTRY   FU       4      HOPE     FL      22
INPFL    AL CR0      40  IRVI     U      10      JI       AL JI
JL0      AL JL0      67  JLI      AL JLI      67  JL2      AL JL2      67
```

```
GRP
NAME      LENGTH RECORD  NAME      LENGTH RECORD  NAME      LENGTH RECORD
APPB     FL 200     APPD     FL 300     BIGEM    FL 500
DIX1     FL 350     DIX2     FL 300     DIX3A    FL 250
DIX4A    FL 300     DIXAP    FL 250     DIXF     FL 150
ED128Z   FL 100     EMI      FL 300     EMS      AL 300      63
EMS1     FL 300     GIN      FL 90
HSI      FL 350     HS11     AL 300      63  LP0      AL LP0      66
LP1      AL LP1      66  MARY     FL 75
RSAM1    FL 300     RSAM2    FL 300
TABLES   FL 200     TERRI    FL 20
SCR      FL 2
```

```
CATALOG-TABLES
NAME      LENGTH RECORD  NAME      LENGTH RECORD  NAME      LENGTH RECORD
TABLES    FL 200     TERRI     FL 20          TLIST     C 1
```

PAPER TAPE

You may choose to save your programs on paper tape as well as in your library or instead of using the library. A set of commands allow you to punch a program from your work space to paper tape, to read the paper tape back into your work space, and subsequently to return control to the keyboard so you can run or modify the program.

PUNCHING THE TAPE. In order to use the PUNCH command, you must have a terminal with an auxiliary paper tape punch. You must first make sure that the paper tape is mounted and that the punch is turned off. (If the punch is on, it will punch any commands you enter as well as the contents of your work space.) Then type PUNCH and *return*. When the system issues two carriage returns, turn on the punch. The contents of your work space will be punched and simultaneously listed at the terminal. Leader is punched before your first statement and again following the last statement. When the punch stops, you should turn it off again or anything else you type will be punched.

If your work space is named, the name is printed before the system issues the carriage returns. Wait until the name is printed before turning on the punch or the name will be punched on your tape.

To illustrate the sequence, suppose you retrieve the program MYPROG from your library, and then punch it on paper tape:

```
GET-MYPROG
PUNCH
MYPROG
```

← *turn on the punch*

```
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

← *turn off the punch*

You now have a copy of MYPROG on paper tape. You may also make tapes of portions of your work space. Refer to the TAPE command, Section X, for further information.

READING THE TAPE. In order to read a paper tape, your terminal must be equipped with a paper tape reader. If it is, place the tape in the reader, turn on the reader, and type the command TAPE. The program is read into your work space and is listed as it is read. If you already have statements in your work space, the tape statements are merged with them. It is, therefore, often necessary to clear your work space with the SCRATCH command before entering TAPE.

To illustrate, read the paper tape of MYPROG:

SCR

←————— *turn on reader*

TAPE

```
10 REM...FIND THE PRODUCT
20 PRINT "ENTER TWO NUMBERS SEPARATED BY A COMMA"
30 INPUT B,C
40 LET A=B*C
50 PRINT A
60 END
```

Note that you should always turn on the paper tape reader before entering the TAPE command. This is important since the system responds to TAPE with a special character used to initiate reading.

After reading the tape, you cannot enter further commands at the keyboard until you enter the KEY command. KEY terminates the effects of TAPE and returns to standard keyboard interaction. If, for instance, you want to run the program you have just read, you must first enter the KEY command:

KEY
RUN

```
ENTER TWO NUMBERS SEPARATED BY A COMMA
? 6,9
  54
```

DONE

MAGNETIC TAPE CARTRIDGES

The same commands that punch to or read programs from paper tape can be used to save and retrieve programs on magnetic tape cartridges provided with the HP 2644 terminal. For any of these operations, you must log on using terminal type 1.

WRITING TO THE CARTRIDGE. Just as with the paper tape, you must have a program or program segment in your work area. You then enter the PUNCH command. If the program has a name, wait for the name to be displayed and then press RECORD on your terminal keyboard; if the work area is unnamed, press RECORD after pressing *return* to enter the PUNCH command.

The contents of the program are not listed at your terminal as they are written. The only indication that the program is being transferred is the green light on the eject button associated with the unit where your cartridge is mounted. The light blinks during the write operation. Only after it has stopped blinking can you assume that your program is completely transferred. When the light has stopped, press *return*. You may now remove the cartridge on which your program is stored.

To summarize this sequence, suppose you want to save MYPROG on a cartridge:

```

HEL-C234, JJ, 1 ← terminal type must be 1
← mount blank cartridge

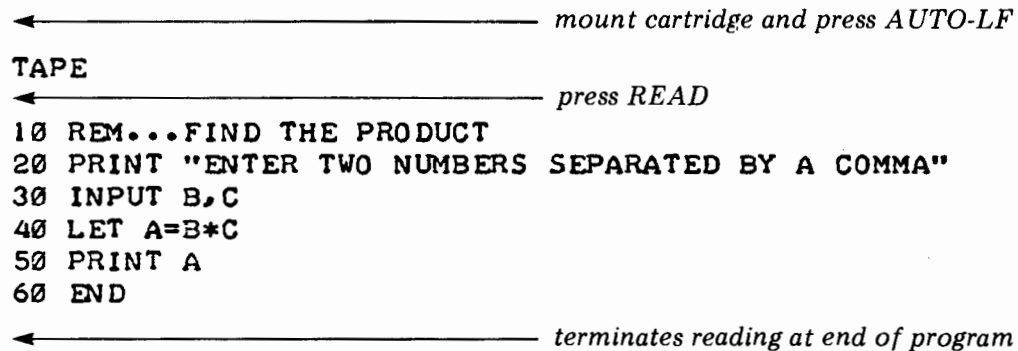
READY
GET-MYPROG
PUNCH
MYPROG
← press RECORD

(green light blinks
while MYPROG is
written to the
cartridge)
return ← press return only after light has stopped blinking

```

READING FROM THE CARTRIDGE. After inserting the cartridge to be read, press down the AUTO-LF button. Then enter the TAPE command and press the READ button. The cartridge is read from a previously selected tape read slot; by default, this is the left hand slot. As the program stored on the cartridge is read into your work space, the statements are displayed at your terminal. You may stop the display at any time by pressing and holding down *return*. You may continue reading by pressing the READ button again. If you do not stop the display, it will continue until the end of the program is reached and then stop automatically.

To summarize the read process:



In order to modify or execute the program, you must return to the keyboard mode by entering the KEY command exactly as if a paper tape had been read.

PROGRAMMING WITH ARRAYS

SECTION

III

WHAT ARE ARRAYS?

An array is a set of variables (or *elements*) which is known by one name. The individual *elements* of an array are specified by the addition of a subscript to the array name. For example, $M(7)$ is the seventh *element* of array M . Arrays and array subscripts are rigorously defined in Section X; this section describes array usage.

REFERENCING ARRAYS

Arrays are referenced by an array variable. The variable can be any single alphabetic character (A through Z). Therefore, you may use up to 26 arrays in a program. The maximum number of data elements allowed in a single array is 5000.

REFERENCING ARRAY ELEMENTS

Arrays have either *one* or *two* dimensions. A *one-dimensional* array consists of a single column of many rows. Individual elements are specified by a single subscript, indicating the row desired. Rows and columns are numbered starting with 1. A *two-dimensional* array consists of a specified number of rows and a specified number of columns organized into a table. For example, an array M of five rows and three columns can be represented as follows:

		Columns		
		1	2	3
Rows	1	$M(1,1)$	$M(1,2)$	$M(1,3)$
	2	$M(2,1)$	$M(2,2)$	$M(2,3)$
	3	$M(3,1)$	$M(3,2)$	$M(3,3)$
	4	$M(4,1)$	$M(4,2)$	$M(4,3)$
	5	$M(5,1)$	$M(5,2)$	$M(5,3)$

Each element of the array is specified by a pair of subscripts separated by commas; the first indicates the row and the second the column.

The remainder of this section deals with the use of arrays. The manipulation of individual array elements is not discussed here since they are treated in the same manner as other numeric variables.

DIMENSIONING ARRAYS

Arrays whose number of rows or columns exceed 10 must be dimensioned using a DIM or COM statement. This allows the system to allocate storage space for the array elements. Arrays whose rows or columns do not exceed 10 do not have to be dimensioned. The system will automatically allocate space (10 elements for one-dimensional arrays, 100 elements for two-dimensional arrays) for undimensioned arrays.

If the array being dimensioned is used in one program, the DIM statement must be included in the program using the array. If the array is to be used in common by more than one program, it must be dimensioned with a COM statement (refer to Section VII for a description of linked programs using data in common). The DIM statement does not have to precede the array reference, but must be in the same program.

The following statement dimensions three arrays:

```
10 DIM A(10,20),B(30),R(30,30)
```

These dimensions specify the *physical size* of each array; the size of array A is 200 elements, of array B 30 elements, and of array R 900 elements. The *logical size* of an array is the current number of rows times the current number of columns. The logical size of an array can be changed by certain statements during program execution. Such new dimensions must never exceed the total physical size and the number of dimensions cannot be changed. That is, array A cannot have more than a total of 200 elements and it cannot be redimensioned as a one-dimensional array with 200 elements. (Refer to REDIMENSIONING ARRAYS, below.)

The DIM statement is used also to dimension strings with more than one character. (Refer to Section IV for a description of strings.) The same statement may be used to dimension both strings and arrays.

PLACING VALUES INTO ARRAYS

The values of array elements are undefined when a program begins execution. If you attempt to reference an undefined array element your program will terminate.

There are several methods of assigning values to arrays. Individual elements can be assigned using the assignment statement:

```
10 LET A[5]=26
15 LET N=15
20 LET B[1,9]=N*4.5
```



In addition, individual elements can appear in INPUT and READ statements.

```
30 INPUT A[1],A[2],A[3]
40 READ B[2,2]
50 DATA 2.5
```

A number of statements allow you to place values into an entire array; three of these are the MAT assignment, MAT INPUT, and MAT READ statements.

The MAT assignment statement copies one array into another. The array on the right is copied into the array on the left. The destination array must have as many elements as the source array, the same number of dimensions, and the same number of elements in each dimension.

The MAT READ statement assigns values from DATA statements to entire arrays, row by row. The MAT INPUT statement is identical to MAT READ except that the values are entered from your terminal as in an INPUT statement. You must continue to enter input for as many elements as the array contains. It is possible with both MAT READ and MAT INPUT to specify new dimensions as parameters to the statements as described below under REDIMENSIONING ARRAYS.

The following short program reads data into array B with MAT READ and then assigns the contents of B to array A with the MAT assignment statement:

```
10 DIM A[2,3],B[2,3]
20 MAT READ B
30 MAT A=B
40 PRINT A[1,1],A[1,2],A[1,3],LIN(2),A[2,1],A[2,2],A[2,3]
50 DATA 2.5,47.7,75,0,50,19.8,0
60 END
RUN
```

```
2.5          47.7          75
0            50           19.8
NONE
```

In the next example, the MAT INPUT statement expects input from the user. Both arrays A and C are printed in their entirety using FOR loops.

(user input is underlined)

```

10 DIM A[3],C[3,2]
20 MAT INPUT A
30 FOR N=1 TO 3
40 PRINT A[N]
50 NEXT N
60 MAT INPUT C
70 FOR M=1 TO 3
80 FOR N=1 TO 2
90 PRINT C[M,N]
100 NEXT N
110 NEXT M
120 END
RUN

```

```

? 27,33,56
  27
  33
  56
? 11,22,33,44,55,66
  11
  22
  33
  44
  55
  66

```

DONE

PRINTING DATA FROM ARRAYS

The methods of printing data from arrays are parallel to those used for filling arrays. Individual elements can be printed using PRINT, but for printing entire arrays the MAT PRINT statement saves you the trouble of entering each element individually.

With the MAT PRINT statement, you can print one or more complete arrays in a single statement. The elements are printed row by row and can be spaced out in fields or packed together, as in the PRINT statement.

Each row of each array is printed separately, with double spacing between rows. If a comma follows the array, each element starts in one of the five divisions of the line. If a semicolon follows the array, the elements are printed packed together, as if each element were followed by a semicolon. If nothing follows the last array in the statement, a comma is assumed. All formatting is done according to the specifications under the PRINT statement.

To illustrate the MAT PRINT statement, this statement is substituted for the PRINT statements in the preceding two examples:

```

10 DIM A[2,3],B[2,3]
20 MAT READ B
30 MAT A=B
40 MAT PRINT A
50 DATA 2.5,47.7,75,0,50,19.8,0
60 END
RUN

```

```

2.5           47.7           75
0             50             19.8

```

DONE

Each row of the array is printed on a separate line with double spacing between rows. If there are more columns than will fit on one line, the line is continued to the next. The next row starts at the beginning of a new line. One-dimension arrays are printed with each element in a new line as illustrated by MAT PRINT A below.

```

10 DIM A[3],C[3,2]
20 MAT INPUT A
40 MAT PRINT A
60 MAT INPUT C
90 MAT PRINT C
120 END
RUN

```

```

?100,.205,35.9
 100

```

```

.205

```

```

35.9

```

```

?10,20,30,40,50,60
 10           20

```

```

 30           40

```

```

 50           60

```

DONE

The next example illustrates the difference between using a semicolon and a comma in the MAT PRINT statement.

```

10 DIM B[3,5],C[2,2]
20 MAT READ B,C
30 MAT PRINT B,C
40 MAT PRINT B;C;
50 DATA 2,4,6,8,0,1,3,5,7,9
60 DATA 1,2,3,4,5,6,7,8,9,0
70 END
RUN

```

```

      2          4          6          8
      1          3          5          7
      1          2          3          4
      6          7
      8          9
      2    4    6    8    0
      1    3    5    7    9
      1    2    3    4    5
      6    7
      8    9

```

DONE

REDIMENSIONING ARRAYS

While the *physical size* of an array cannot be changed during execution, the *logical size* can be changed by entering a *new dimensions* parameter with some statements. Those statements that allow you to enter new array dimensions are MAT INPUT, MAT READ, and three statements with which you can initialize arrays: MAT...ZER, MAT...CON, and MAT...IDN. Each of these statements may be followed by the new dimensions in parentheses and separated by a comma if more than one. The number of dimensions cannot be changed.

To illustrate, array A in the following example is redimensioned by a MAT READ statement:

```

10 DIM A(30,20)          physical size is 600 elements
.
.
.
50 MAT READ A(600,1)    physical size is still 600 elements

```

The new dimensions effectively make array A a one-dimensional array, but it is illegal to actually change the number of dimensions. For instance, the statement 50 MAT READ A(600) would be diagnosed as an error since the array was originally dimensioned with both rows and columns.

In the following example, array B is redimensioned in the MAT INPUT statement; only the number of elements specified in the new dimensions may be input when the program is executed.

```

10 DIM A[10,4]
20 MAT INPUT A[2,2]
30 MAT PRINT A
40 END
RUN

```

```

? 123.45, 325.67, 456.22, 438.56
  123.45          325.67

  456.22          438.56

```

DONE

Note that the statement MAT INPUT A(2,2) requests four elements, whereas the statement INPUT A(2,2) asks for only one element of array A.

INITIALIZING ARRAYS

Three special functions (MAT...ZER, MAT...CON, and MAT...IDN) provide the means to initialize arrays with certain values, and, optionally, to redimension the arrays.

MAT...ZER sets all elements of the array to zero. MAT...CON sets all elements of the array to one. MAT...IDN assigns an identity array to the array specified. The identity array is all zeros, except the major diagonal (top left corner to bottom right corner), which is all ones.

If an array is redimensioned by MAT...ZER, MAT...CON, or MAT...IDN, the new size cannot have more elements than the physical size, nor can the number of dimensions be altered. MAT...IDN must always be dimensioned with the same number of rows as columns.

In the following example, MAT...ZER sets each element of array A to zero:

```

10 DIM A[2,3]
20 MAT A=ZER
30 MAT PRINT A
40 END
RUN

```

```

  0          0          0
  0          0          0

```

DONE

Programming with Arrays

MAT A = CON(3,4) redimensions array A to have 3 rows and 4 columns, and sets each element in the newly-dimensioned array to 1.

```
10 DIM A[4,4]
20 MAT A=CON[3,4]
30 MAT PRINT A
40 END
RUN
```

```
1      1      1      1      1
1      1      1      1      1
1      1      1      1      1
```

DONE

MAT A = IDN(4,4) changes the dimensions of A to 4 rows by 4 columns and sets the major diagonal to 1, the remaining elements to 0.

```
10 DIM A[5,5]
20 MAT A=IDN[4,4]
30 MAT PRINT A
40 END
RUN
```

```
1      0      0      0
0      1      0      0
0      0      1      0
0      0      0      1
```

DONE

ARRAY OPERATIONS

The following group of five statements provides functions which operate on one or more entire arrays:

- MAT Addition and Subtraction statement
- MAT Multiplication statement
- MAT ... INV (Inverse) statement
- MAT ... TRN (Transpose) statement
- MAT Scalar Multiplication statement

ARRAY ADDITION/SUBTRACTION

The MAT addition and subtraction statement performs array addition or subtraction (element by element) upon arrays of identical dimensions and assigns the result to another array.

Examples:

```
10 DIM A(2,2),B(2,2),C(2,2)
20 MAT READ A,B
30 MAT C=A+B
40 MAT PRINT A,LIN(2),B,LIN(2),C
50 DATA 3,3,4,4,2,2,1,1
60 END
RUN

      3          3
      4          4

      2          2
      1          1

      5          5
      5          5

DONE
```

Programming with Arrays

The values in arrays A and B are added to produce the values printed for array C. Using the same data, A is subtracted from B to produce the following results in C:

```
10 DIM A(2,2),B(2,2),C(2,2)
20 MAT READ A,B
30 MAT C=A-B
40 MAT PRINT A,LIN(2),B,LIN(2),C
50 DATA 3,3,4,4,2,2,1,1
60 END
RUN
```

```
3          3
```

```
4          4
```

```
2          2
```

```
1          1
```

```
1          1
```

```
3          3
```

```
DONE
```

ARRAY MULTIPLICATION

The MAT Multiply statement performs an array multiplication on an array of dimension m by n and an array of dimension n by p ; that is, the number of columns in the first array must equal the number of rows in the second. The result, a new array of dimension m by p , is assigned to a third array.

Each row of the array to the left of * is multiplied by each column of the array on the right to produce the new element. The resulting array is assigned to the array to the left of the assignment operator. This array is redimensioned to dimension m by p . The same array may not appear on both sides of the equal sign.

Examples:

```

10 DIM A[2,3],B[3,2],C[2,2]
20 MAT READ A,B
30 MAT C=A*B
40 MAT PRINT A;LIN(1);B;LIN(1);C;
50 DATA 1,2,3,4,5,6
60 DATA 4,5,6,7,8,9
70 END
RUN

```

1	2	3	}	array A
4	5	6		

4	5	}	array B
6	7		
8	9		

40	46	}	array C = A*B
94	109		

DONE

The method for performing a matrix multiplication is to multiply each element of the first row of array A by the corresponding element of the first column of B and to add the products. The result is the element C(1,1). Then each element in the first row of A is multiplied by the corresponding element in the second column of B and these are added to produce C(1,2). C(2,1) is the sum of the products resulting from the multiplication of row 2 of A and column 1 of B; C(2,2) is the sum of the products of row 2 of A and column 2 of B. To illustrate:

$$1 \times 4 (4) + 2 \times 6 (12) + 3 \times 8 (24) = 40$$

$$1 \times 5 (5) + 2 \times 7 (14) + 3 \times 9 (27) = 46$$

$$4 \times 4 (16) + 5 \times 6 (30) + 6 \times 8 (48) = 94$$

$$4 \times 5 (20) + 5 \times 7 (35) + 6 \times 9 (54) = 109$$

A second example multiplies the square array C by itself. In this case, the number of columns always equals the numbers of rows.

(user input is underlined>)

```

10 DIM C[3,3],D[3,3]
20 MAT INPUT C
30 MAT PRINT C;
40 MAT D=C*C
50 MAT PRINT D;
60 END
RUN

```

```

? 2,4,6,8,1,3,5,7,9
  2     4     6
  8     1     3
  5     7     9
 66    54    78
 39    54    78
 111   90   132

```

DONE

To achieve the result MAT D=C*C;

$$D(1,1) = 2 \times 2 (4) + 4 \times 8 (32) + 6 \times 5 (30) = 66$$

$$D(1,2) = 2 \times 4 (8) + 4 \times 1 (4) + 6 \times 7 (42) = 54$$

$$D(1,3) = 2 \times 6 (12) + 4 \times 3 (12) + 6 \times 9 (54) = 78$$

$$D(2,1) = 8 \times 2 (16) + 1 \times 8 (8) + 3 \times 5 (15) = 39$$

$$D(2,2) = 8 \times 4 (32) + 1 \times 1 (1) + 3 \times 7 (21) = 54$$

$$D(2,3) = 8 \times 6 (48) + 1 \times 3 (3) + 3 \times 9 (27) = 78$$

$$D(3,1) = 5 \times 2 (10) + 7 \times 8 (56) + 9 \times 5 (45) = 111$$

$$D(3,2) = 5 \times 4 (20) + 7 \times 1 (7) + 9 \times 7 (63) = 90$$

$$D(3,3) = 5 \times 6 (30) + 7 \times 3 (21) + 9 \times 9 (81) = 132$$

This next example multiplies a two-dimensional array with three rows and two columns by a one-dimensional array with two rows. The result is a one-dimensional array with three rows.

```

10 DIM A[3,2],B[2],C[3]
20 MAT READ A
30 MAT READ B
40 MAT C=A*B
50 DATA 1,2,3,4,5,6,1,2
60 MAT PRINT A;LIN(1);B;LIN(1);C;
70 END
RUN

```

```

1    2
3    4
5    6

```

```
1
```

```
2
```

```
5
```

```
11
```

```
17
```

```
DONE
```

To achieve the result $\text{MAT C}=\text{A}*\text{B}$:

$$C(1) = 1 \times 1 (1) + 2 \times 2 (4) = 5$$

$$C(2) = 3 \times 1 (3) + 4 \times 2 (8) = 11$$

$$C(3) = 5 \times 1 (5) + 6 \times 2 (12) = 17$$

ARRAY INVERSION

The MAT . . . INV statement assigns the inverse of a square array (i.e., number of rows equals number of columns) to another array. The inverse of an array is the array which, when multiplied by the original array, results in the identity array.

The two arrays must be of the same dimensions. The same array may be used on both sides of the equation.

Example:

(user input is underlined)

```
10 DIM A(5,5),B(5,5)
20 MAT INPUT B
30 MAT A=INV(B)
40 MAT PRINT B,LIN(2),A
50 END
RUN
```

```
? 1,0,0,0,0,2,1,0,0,0,3,2,1,0,0,4,3,2,1,0,5,4,3,2,1
  1      0      0      0      0
  2      1      0      0      0
  3      2      1      0      0
  4      3      2      1      0
  5      4      3      2      1

  1      0      0      0      0
-2.      1.      1.78814E-07      0      0
  1.      -2.      1.      0      0
  7.74860E-07      .999999      -2.      1      0
-1.01328E-06      7.15256E-07      1.      -2      1
```

DONE

25 values are input to the square array B, then using INV, array A is set to the inverse of B.

ARRAY TRANSPOSITION

The MAT . . . TRN statement assigns the transposition of an n by m array to an m by n array. Transposition switches rows and columns.

The array on the left is redimensioned such that the resulting array is the reverse of the original array. The same array cannot be used on both sides of the equation.

Example:

(user input is underlined)

```
10 DIM A[5,3],B[3,5]
20 MAT INPUT B
30 MAT A=TRN(B)
40 MAT PRINT B,LIN(2),A
50 END
RUN
```

```
? 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  1           2           3           4           5
  6           7           8           9          10
 11          12          13          14          15

  1           6          11
  2           7          12
  3           8          13
  4           9          14
  5          10          15
```

DONE

Array A is the result of transposing array B with the TRN function. The columns in B are the rows in A; the rows in B are the columns in A.

ARRAY SCALAR MULTIPLICATION

The MAT scalar multiplication statement multiplies all of the elements of an array by a specified value and assigns the result to another array.

Example:

(user input is underlined)

```

10 N=5
20 MAT INPUT B
30 MAT A=(N*2)*B
40 MAT PRINT A;
50 DIM A[3,4],B[3,4]
60 END
RUN

```

```

? 1,2,3,4,5,6,7,8,9,10,11,12
  10    20    30    40

  50    60    70    80

  90   100   110   120

```

DONE

Scalar multiplication simply multiplies each element of the array by the specified *numeric expression*, in this case $N*2$ or 10 since $N=5$. Each element of the resulting array A is 10 times the corresponding element in B. The dimensions of A are copied from B. The *numeric expression* must be enclosed in parentheses.

WHAT ARE STRINGS?

Strings are groups of characters which can be used to provide program dialog with the user to facilitate data input or to print comments and headings during report generation. Strings are the principal type of data used in applications such as text editing and most data base applications. For example, the following statement could be used to print a heading in a report generation program:

```
10 PRINT "QUARTERLY REPORT"
```


string

STRING CHARACTER SET

The characters used in strings must be from the ASCII (American Standard Code for Information Interchange) character set. The full set of characters available is given in Appendix A. The character set includes upper and lower case letters, numbers, punctuation, and a variety of non-printing characters (used to control input/output devices). This means that commas, periods, and control characters such as *return* and *line feed* are valid string characters. Strings are normally enclosed in quotes ("THIS IS A STRING"). Examples of strings are:

```
"ABCDEFGHI"
```

```
"12345"
```

```
"Bob and Tom"
```

```
"March 15, 1970"
```

```
" " (Quotes without an enclosed character define the 'null' string)
```

NUMERIC EQUIVALENTS OF CHARACTERS (AN ALTERNATE FORM)

Strings can also be represented as the numeric equivalent of a character preceded by an apostrophe. The numeric equivalent can be any integer from 0 to 255. A complete list of numeric equivalents of characters is given in Appendix A. The numeric equivalent of "A" for example, is 65. The character "A" can therefore be represented as '65. In a similar manner, a *line feed* is '10. Numeric equivalents exist for all ASCII characters including quotes, H^c, and X^c.

Examples:

'23 '64 '49 is equivalent to "W^c@1"
'65 '66 '67 is equivalent to "ABC"

The numeric equivalent form of string character can be combined with other string characters to form longer strings.

Example:

'65 "BC" '68 '69 "F" is equivalent to "ABCDEF"

When assigning values to strings, two quoted strings cannot be adjacent ("ABCD" '69 is valid, "AB" "CD" '69 is not). The length of the combined string cannot exceed the string character limit of 255. Note that the numeric equivalent form of string characters cannot be used to respond to input statements (INPUT, LINPUT, LINPUT#, ENTER, or READ#). When you list a program that has non-printing characters within a quoted string, the numeric equivalent of the non-printing character is printed.

UPPER AND LOWER CASE LETTERS

Lower case letters can be input from or output to terminals having a lower case capability. When lower case letters are output to a terminal not capable of printing them, the terminal normally prints the upper case equivalent. Lower case letters are automatically converted to upper case by the system, except when they occur in strings or REM statements. Lower case letters in file names used in FILES, ASSIGN, CREATE, and PURGE operations or program names used in CHAIN statements are converted to upper case when used as are format specifications appearing in strings referenced by PRINT USING statements.

HOW DO YOU REFERENCE STRINGS?

Strings can be used directly as in the statement `10 PRINT "ENTER DATA"`, or they can be given a name and then the name can be referenced whenever the string of characters is required.

NAMING STRINGS

String names (variables) are made up of a single alphabetic character (A through Z) followed by \$, 0\$, or 1\$. This allows up to 78 different string variables to be defined.

Examples:

```
A$, B0$, C1$
```

DIMENSIONING STRINGS

String variables are assumed to be one character long by default. If a string will exceed one character it must be dimensioned in a DIM or COM statement. This ensures that sufficient program space is allocated. Attempts to create a string longer than its dimensioned value will result in an error. Strings of less than the dimensional length are valid.

When a string variable is declared, its "physical" length is set. The "physical" length is the maximum size string that the variable can accommodate. The maximum physical length accepted by 2000 BASIC is 255 characters.

Examples:

```
710 DIM A$(72),B$(20),C$(50),Z1$(255)
720 COM R$(200),S0$(10)
```

During execution of a program, the "logical" length of a string variable may vary. The "logical" length of the variable is the actual number of characters that the string variable contains at any point.

Examples:

```
100 DIM A$(72)    (Sets physical length of A$ to 72)
200 A$ = "SAMPLE STRING"    (Logical length of A$ is 13)
300 A$ = "LONGER SAMPLE STRING"    (Logical length of A$ is now 20)
```

SUBSTRINGS

Substrings are contiguous subsets of string characters. The subset is defined with subscripts following the string variable name. Two subscripts, separated by a comma, specify the first and last characters of the substring. Characters within a string are numbered from the left starting with one. Subscripts must be positive, non-zero and less than 32,768. (Note that even though subscripts up to 32,768 are allowed, execution errors result when subscripts exceed the string size defined in a DIM statement or 255 characters.) Non-integer subscripts are rounded to the nearest integer.

Example:

```

95  DIM Z$(10)
100 Z$="ABCDEFGH"
200 PRINT Z$(2,6)

```

prints the substring

BCDEF

A single subscript specifies the first character of the substring and implies that all characters following are part of the substring. Continuing the example:

```

300 PRINT Z$(3)

```

prints the substring

CDEFGH

Two equal subscripts specify a single character substring.

```

400 PRINT Z$(2,2)

```

prints the substring

B

If a substring is specified which is larger than the physical length of the original string, blanks are appended to the substring in place of the missing characters.

HOW DO YOU USE STRINGS?

Strings may be used in a variety of ways. The first step is to assign a value to the string. Once this has been done there are a variety of operators and string functions that can be used to test or modify the string. When string modification is complete, the string can be printed in the same manner as numeric data.

PLACING VALUES IN STRINGS

You assign values to string variables by setting them equal to string data, other strings, or string valued functions.

SIMPLE ASSIGNMENT. The simplest way to set a string value is to use the LET statement. Strings can be set equal to literal strings, other string variables, or substrings.

Examples:

```

5  DIM X0$(16),R$(5),G$(10)
10 LET X0$="THIS IS A STRING"
15 G$="1234567890"
20 R$(1,5)=G$(6,10)
25 PRINT X0$,R$(1,5)
30 END
RUN

```

```

THIS IS A STRING          67890

DONE

```

In the first example, X0\$ is assigned the value in quotes. In the second example, the value contained in the substring G\$(6,10) is assigned to the substring R\$(1,5).

STRING DATA. String variables can be set equal to string data using the INPUT, LINPUT, ENTER, or READ statements. The first three statements are used to set a string or substring equal to string data entered from your terminal.

Examples:

```

5  DIM A$(5),B0$(5),C$(10),X$(72)
7  E=30
10 INPUT A$ ← (Inputs a string value from your terminal and assigns it to A$)
20 INPUT B0$,C$(1,5) (Inputs a string value and assigns it to B0$ and inputs a second
                    string value and assigns it to the first five characters in C$)
30 LINPUT X$ (Inputs an entire line from your terminal and assigns it to X$)
40 ENTER #V,E,A,A$ (Inputs a string value from your terminal and assigns it to A$)
50 END
RUN

```

```

?ID=
?X.K73
??A.396
THE NEXT LINE MUST BE ENTERED IN 30 SECONDS:
W.X93
DONE

```

The READ statement can be used to set a string or substring equal to string data contained in a DATA statement. Strings used in DATA statements are limited to a maximum of 255 characters. (This may be less depending on your system's configuration.)

Example:

```

5 DIM A$(3),B$(3),C$(3),D$(10)
7 D$=" TOO "
10 READ A$,B$,C$,N,D$(6,8)
20 DATA "ABC","BCA","CAB",5,"BAD"
30 PRINT A$,B$,C$,N,D$(1,8)
40 END
RUN

```

```

ABC          BCA          CAB          5          TOO BAD

DONE

```

The example assigns each of the values given in the DATA statement to the corresponding variable in the READ statement. Note that string and numeric variables can occur in the same READ statement.

The READ# and LINPUT# statements can be used to obtain values from a file rather than your terminal. (Refer to Section V for a discussion of file operations.)

Examples:

```

10 FILES FILE1,FILE2
20 N=2
30 DIM A$(10),B$(30),C$(7),G$(100)
40 READ #1;A$,B$,C$(1,7)
50 LINPUT #N;G$
60 END

```

SETTING STRINGS EQUAL TO STRING VALUED FUNCTIONS. String valued functions (refer to the discussion following) can be used in place of string data in assignment operations. The string variable is set equal to the result of the function.

Examples:

```

10 DIM A$(5),B$(5),Q$(3)
20 READ B$,N
30 A$=UP$(B$)
40 Q$(1,1)=CHR$(N)
50 PRINT A$,Q$(1,1)
60 DATA '97'98'99'100'101,65
70 END
RUN

```

```

ABCDE          A

DONE

```

USING STRINGS IN RELATIONAL OPERATIONS

Strings and string variables can be used with relational operators in the same manner as numeric values. The relational operators (=, <, >, <=, >=, <>) can be used with strings to perform branching operations. These operators are also useful for sorting strings.

Example:

```

10 DIM A$(3),B$(7),C$(3)
20 B$="XYZABCD"
30 INPUT A$
40 IF A$ >= B$(3,7) THEN 80
50 INPUT C$
60 IF C$ <> "ABC" THEN 80
70 PRINT "A$ < ZABCD AND C$ = ABC"
80 END
RUN

```

```

?123
?ABC
A$ < ZABCD AND C$ = ABC

DONE

```

STRING STATEMENTS AND FUNCTIONS

There are several statements and functions that perform special string operations. Some use a string or numeric argument and return string values, others return numeric values.

STRING VALUED STATEMENTS AND FUNCTIONS. The statements and functions that return string values are the CONVERT statement and the CHR\$, and UPS\$ functions.

The CONVERT statement allows you to change numeric data to its ASCII equivalent and an ASCII number to its numeric equivalent. (Note, this is not the equivalent of individual characters as in CHR\$ or NUM.) In the following example the value of N is converted to the string of ASCII characters that would be listed at your terminal by the LIST command.

Example:

```

5 DIM A$(3)
10 N=2*(111)
20 CONVERT N TO A$
25 PRINT A$
30 END
RUN

```

```

222

```

```

DONE

```

A\$ now contains the ASCII characters “^222”. (The leading blank would be a minus sign if N were negative.) The ASCII string representation can now be used in alphanumeric sort operations or can be appended to other strings.

For example, if N were used as a counter for labeling files, a new file name (string) could be created by appending the counter value (number) each time a new file is required.

Example:

```

5  DIM F$(6)
10  F$="FIL"
20  N=0
30  GOSUB 100
   :
   :
80  STOP
90  REM FILE CREATOR
100 N=N+1
110 CONVERT N TO F$(4)
120 PRINT "CREATE FILE "F$
130 CREATE R,F$,100
140 RETURN
150 END
RUN

```

CREATE FILE FIL1

DONE

The first time the subroutine is entered, a file name FIL1 containing 100 records would be created. Additional passes through the subroutine would create the files FIL2, FIL3, FIL4, etc.

The CHR\$ function converts a number in the range $0 \leq n \leq 255$ into the equivalent ASCII character. (This is the opposite of the NUM function which returns a numeric equivalent for a string character.) A complete list of ASCII characters and their numeric equivalents is given in Appendix A.

Example:

```

10  DIM A$(255)
20  FOR N=1 TO 255
30  A$(N)=CHR$(N)
40  NEXT N
50  PRINT A$(65,90)
60  END
RUN

```

ABCDEFGHIJKLMN OPQRSTUVWXYZ

DONE

This example would create a string (255 characters long) where each character is the ASCII equivalent of its position in the string. In other words $A\$(65) = "A"$, $A\$(66) = "B"$, $A\$(67) = "C"$, etc.

The `UPS$` function causes lowercase alphabetic characters (a through z) to be shifted to their uppercase equivalents (A through Z).

Example:

```
10 A$ = UPS$(Z$)
```

The `UPS$` function is valuable in alphanumeric sorts. Sorting is normally done according to the ASCII equivalent value of each character. If the strings to be sorted contain a mix of upper and lowercase characters it is possible that an "a" would be set after "Z". To eliminate this problem you can shift strings to their uppercase equivalent before sorting. Once the sort position is obtained the lowercase string can be entered in the proper table position.

NUMERIC VALUED STATEMENTS AND FUNCTIONS. The statements and functions that use string arguments and return numeric values are the `CONVERT` statement and the `LEN`, `NUM`, and `POS` functions.

The `CONVERT` statement allows you to change an ASCII number to its numeric data equivalents and numeric data to its equivalent in ASCII characters. (Note, this is not the same as the equivalence of individual characters as in `CHR$` and `NUM`.) In the following example the value of `A$` is converted to a numeric data value and assigned to `N`. This is just the opposite of the example used earlier with the `CONVERT` statement.

Example:

```
5  DIM A$(5)
10 A$="995.6"
20 CONVERT A$ TO N
25 PRINT "N="N
30 END
RUN
```

```
N= 995.6
```

```
DONE
```

The numeric variable `N` now contains the numeric value 995.6. This numeric value can now be used in numeric expressions. This conversion is useful when formatting reports involving arithmetic operations and textual data.

The `LEN` function returns a numeric value that is the length (number of characters) currently assigned to a string variable. In the example that follows, `N` is set to the number of characters assigned to (not dimensioned for) `A$`.

Example:

```

10 DIM A$(72)
20 A$="ABC"
30 N=LEN(A$)
35 PRINT "N="N
40 END
RUN

N= 3

DONE

```

This would result in N being set to 3. The LEN function is useful in sorting or searching strings or in appending or modifying strings. The example given for the NUM function illustrates how the LEN function can be used to index character by character through a string.

The NUM function converts a single string character to a numeric value that is its ASCII code equivalent. A complete list of ASCII characters and their numeric code equivalent is given in Appendix A. In the following example, the numeric variable N is set to the ASCII code equivalent of the third character in A\$.

Example:

```

10 DIM A$(6)
20 A$="STRING"
30 N=NUM(A$(3,3))
40 PRINT N
45 END
RUN

82

DONE

```

If the NUM argument is more than one character in length, only the first character is converted.

You can obtain the numeric code equivalent of a whole string as follows:

```

5 DIM A$(50),N(50)
7 A$="STRING OF 50 CHARACTERS MAXIMUM"
10 FOR I=1 TO LEN(A$)
20 N(I)=NUM(A$(I,I))
25 PRINT N(I);
30 NEXT I
35 END
RUN

```

```

83      84      82      73      78      71      32      79      70      32      53      48
32      67      72      65      82      65      67      84      69      82      83      32
77      65      88      73      77      85      77
DONE

```

The POS function allows you to determine if one string occurs within another string. This is a powerful tool in any text editing or other word processing application. The POS function uses two string arguments, a string to search and a string to look for. If the "looked for" string is found, the function is set to the position in the "searched" string where the first matched character occurs. If the "looked for" string does not occur, the function is set to 0. In the following example, S\$ is a long string that may contain one or more occurrences of another string R\$.

Example:

```

10 DIM S$(60),R$(10)
20 S$="THIS IS THE POINT TO INPUT THE DATA"
25 R$=" THE "
30 REM P IS THE POSITION IN S$ WHERE R$ BEGINS
35 P=0
40 P1=POS(S$(P+1),R$)
50 IF P1=0 THEN 80
55 P=P1+P
60 PRINT "SEARCH STRING FOUND AT";P
70 GO TO 40
80 END

```

RUN

```

SEARCH STRING FOUND AT 8
SEARCH STRING FOUND AT 27

```

DONE

PRINTING STRINGS

Strings can be printed at your terminal, written to disc files, or sent to any of the peripheral devices available on your system. Output of string data is accomplished in the same manner as numeric data. String and numeric data can be mixed in the same output statement. Printing is performed using the PRINT and PRINT # statements.

The PRINT statement allows you to output string data to your terminal.

Example:

```

90 DIM A$(20),B$(20),C$(20)
95 READ A$,B$,C$
100 PRINT "THIS IS A STRING ";
120 PRINT A$,B$,C$
130 DATA "ASTRING","BSTRING","CSTRING"
150 END
RUN

```

```

THIS IS A STRING  ASTRING      BSTRING      CSTRING

```

DONE

Programming with Strings

The PRINT USING statement can be used to output string data according to a predetermined format. In the following example, the variables A\$, I, and T(I) are printed using the format defined in statement 80. Quoted strings and string-valued functions cannot be used as print items in a PRINT USING statement.

Example:

```
10 DIM A$(10), B$(20)
20 A$="ACCOUNT"
30 B$="TOTAL REPORT"
40 PRINT TAB(10), B$(7, 12)
45 T=0
50 FOR I=1 TO 10
60 READ T(I)
70 PRINT USING 80; A$, I, T(I)
80 IMAGE 7A, DD, 5X, 6D. D
85 T=T+T(I)
90 NEXT I
95 PRINT
100 PRINT USING 105; B$(1, 5), T
105 IMAGE 2X, 5A, 7X, 6D. D
110 DATA 15, 150, 2000, 3000, 600, 50, 70, 19.5, 100, 120
120 END
RUN
OK
```

	REPORT	
ACCOUNT 1		15.0
ACCOUNT 2		150.0
ACCOUNT 3		2000.0
ACCOUNT 4		3000.0
ACCOUNT 5		600.0
ACCOUNT 6		50.0
ACCOUNT 7		70.0
ACCOUNT 8		19.5
ACCOUNT 9		100.0
ACCOUNT 10		120.0
TOTAL		6124.5

Refer to Section XI for further information on the PRINT USING and IMAGE statements.

The PRINT# statement can be used to output data to BASIC formatted or ASCII files (refer to Section V for a discussion of file operation).

Example:

```
10 FILES AA, BB
20 DIM A$(3), B$(10)
30 A$="ABC"
40 B$=" STRING "
50 N=R=2
70 PRINT #1; A$, N, B$, "OUTPUT"
80 PRINT #N, R+1; A$, B$(1, 7), "TEST"
90 END
```

WHAT ARE FILES?

A file is a collection of data organized into records and stored on a device external to your work space such as a disc, paper tape, magnetic tape or a deck of cards.

The data is entered into a file through the HP 2000 file print statements. The data must have been previously entered into your work space or it must be present on another file from which it can be transferred. The process of entering data into files is called *writing to* the file.

The reverse operation, *reading from* a file, is performed by the HP 2000 file read statements. This process retrieves the data from the file and makes it available to your program.

Data may also be transferred from one file to another. For instance, data stored on magnetic tape can be transferred or copied to a disc file by statements that read from the magnetic tape file and then write to the disc file.

The 2000 System has two types of files: BASIC formatted files and ASCII files.

- BASIC formatted files are always on discs and are used for storing large amounts of data such as lists of employees or company inventories.
- ASCII files can be stored on paper tape, cards, discs, and magnetic tape — they are frequently referred to as device files. ASCII files are most useful for directing file data to and from other devices on the system — e.g. to print data on a line printer, or to read data from a deck of cards, or to input data from a link terminal. They can also be used to store and access information on discs.

In this section we will look at how each type of file works and how you can use both types of files in your programs to handle data input, output, and storage.

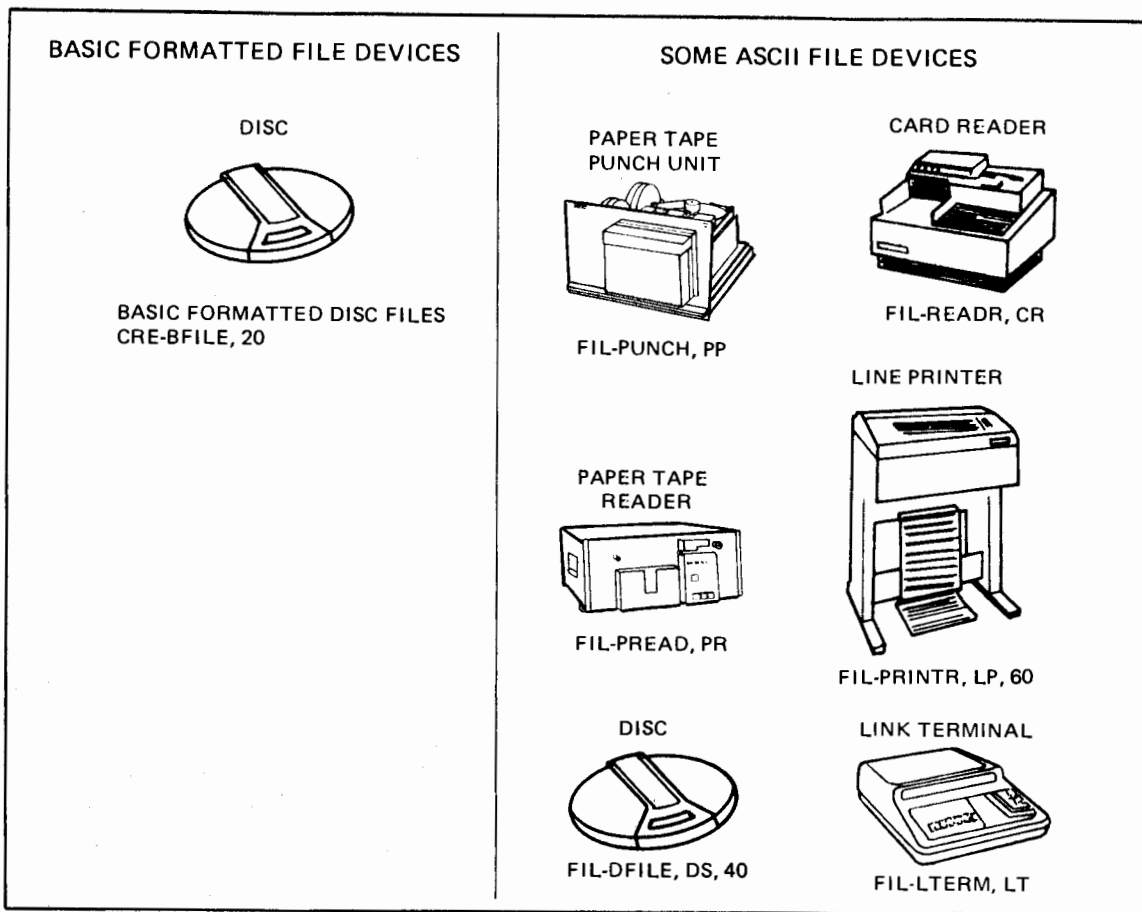


Figure 5-1. BASIC Formatted and ASCII File Devices

BASIC FORMATTED FILES

BASIC formatted files are used primarily to store data for later retrieval and manipulation. You have already seen how data can be stored within a program using DATA statements. For example, the following program prints out the months of the year by reading them from two data statements:

```

10 DIM A$(40)
20 PRINT "THE MONTHS OF THE YEAR:"
30 FOR I=1 TO 12
40 READ A$
50 PRINT A$
60 NEXT I
70 DATA "JANUARY", "FEBRUARY", "MARCH", "APRIL", "MAY", "JUNE", "JULY"
80 DATA "AUGUST", "SEPTEMBER", "OCTOBER", "NOVEMBER", "DECEMBER"
90 END

```

BASIC formatted files are similar to DATA statements. For example, each can store both numeric and string data; each has associated with it a pointer which is initially set to the first item of data and which advances sequentially through the data as items are read; each can be read serially and directly; and for each you can detect whether the next item to be read is a number, a string, or if there is no more data to be read.

DATA statements, although useful in many cases, are not very powerful for two reasons. (1) The amount of data which can be stored in any one program is limited to the maximum size of a program (about ten thousand words of storage, or the equivalent of about twenty thousand characters of data). (2) The data stored in DATA statements never changes from one RUN of the program to the next.

BASIC formatted files overcome these deficiencies. They may be used to store large amounts of data (a BASIC file may contain up to 16.7 million characters of data, depending on the system's configuration), and this data may be added to, changed, and deleted by running programs.



CREATING BASIC FORMATTED FILES

When data is entered in a BASIC formatted file it is organized on the disc using *records* and *words*. Before you can store data in a BASIC formatted file, space for that file must be reserved on a disc using either the CREATE statement or the CREATE command. Each permits you to specify the three defining characteristics of a file:

- name — 1 to 6 letters or digits; must be unique from all other names in your library
- length in records — can be up to 32,767 records long depending on the size of your system disc; automatically numbered consecutively from record 1
- size of each record (optional) — all records physically occupy 256 words of disc storage although for special programming purposes they may be set to a logical length from 64 to 256 words (default = 256 words)

For example, to create a file to store a list of 1000 employees names with their ID numbers and salaries, use the command

```
CRE-SALARY,59
```

This command reserves 59 consecutive records on the disc for a file named SALARY. Since no value is indicated for record size, the default value, 256 words, is assumed. End-Of-File marks are printed in the first word of every record and at the end of the file. When you enter data in a record, the data is written over the beginning EOF (eliminating that EOF). The EOF at the end of the file cannot be overwritten by data.

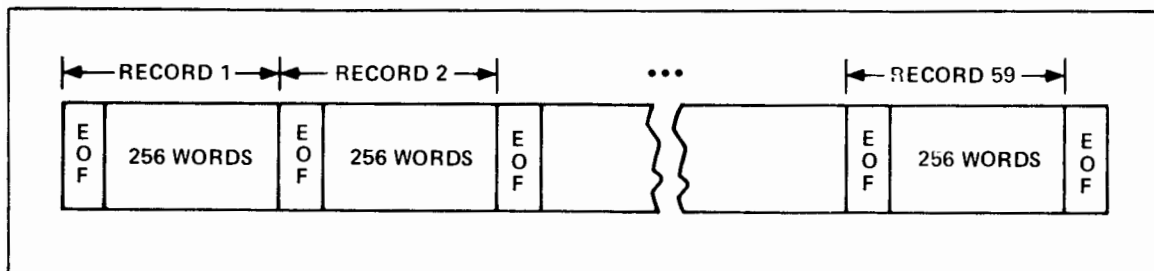


Figure 5-2. Record Format in BASIC Formatted Files

Files

To determine how long the file needs to be, you must first estimate how many words of record space each of the 1000 entries will need. Every string with an even number of characters occupies $(n/2 + 1)$ words, that is, 2 characters per word plus 1 word for the length of the string. Every string with an odd number of characters occupies $((n + 1)/2 + 1)$, that is, 2 characters per word including a full word for the last (odd) character plus 1 word for the length of the string. Each entry in the file SALARY will be of the form: ID, name, salary. Allowing for a maximum length of 20 characters per employee name, each entry will occupy at most 15 words of space in a record (2 words for the ID number, 2 words for the salary, and 11 words for the 20 character name string). Each record (by default) can store 256 words. Data items cannot be split up. This means you cannot put part of an item at the end of one record and the rest of the item at the beginning of the next record. Therefore, each record can hold 17 entries:

$$256 \text{ words} \div 15 \text{ words/entry} = 17 \text{ entries, with 1 word left over}$$

Since the file SALARY is to hold data for 1000 employees and 17 entries fit in a record, 59 records should be reserved for the file:

$$1000 \text{ entries} \div 17 \text{ entries/record} = 59 \text{ records}$$

It is possible that the data will actually occupy fewer than 59 records of storage. If an employee name is shorter than 20 characters then that entry will occupy fewer than 15 words in the record. In the samples below, the first entry would occupy 15 words but the second would occupy only 10 words:

ID	NAME	SALARY
0132	JONATHAN FREDRICKSON	175.00
$\underbrace{\hspace{1.5cm}}$ 2 words	$\underbrace{\hspace{3.5cm}}$ 11 words	$\underbrace{\hspace{1.5cm}}$ 2 words
ID	NAME	SALARY
1408	JANE SMITH	190.00
$\underbrace{\hspace{1.5cm}}$ 2 words	$\underbrace{\hspace{2.5cm}}$ 6 words	$\underbrace{\hspace{1.5cm}}$ 2 words

It may be possible then to fit more than 17 entries in a record. However, you should still reserve enough space for the largest possible case — 1000 entries where every name is 20 characters long. Also, if you do not presently have 1000 employees for the file but anticipate eventually adding new employees, it would be wise, when you initially create the file, to reserve sufficient space to extend the list.

Figure 5-3 shows how the file SALARY, with 1000 data entries, all of maximum length (15 words) is arranged on the disc:

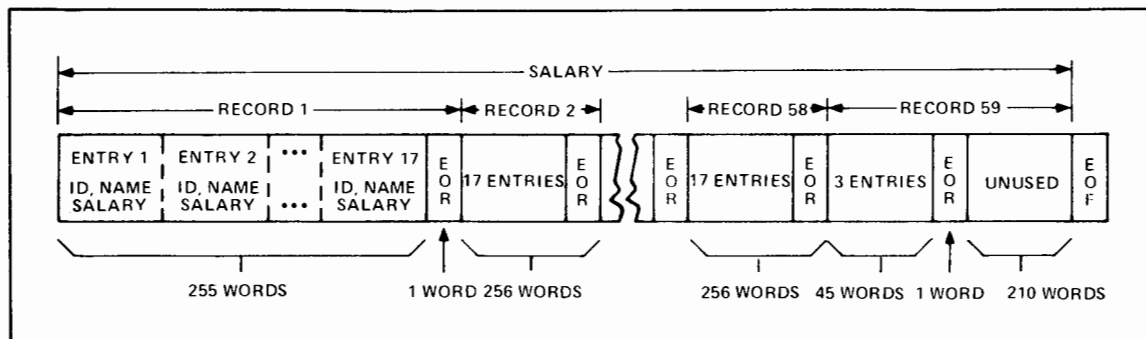


Figure 5-3. Sample File Organization

The data in each record is followed by an end-of-record (EOR) mark and the last record of the file is followed by an end-of-file (EOF) mark. Note that the extra word in each record is used for the EOR mark. The EOF mark follows the last record.

Once created, the file is cataloged in your library. You may get a list of all your files by using the CATALOG command. The file remains on the disc until it is eliminated with a PURGE statement or command.

OPENING AND CLOSING A BASIC FORMATTED FILE

In order to use a file in a program you must first 'open' the file with a FILES statement or an ASSIGN statement. Opening a file causes the file to be associated with an integer *file number* from 1 to 16. Once a file is opened, all references to it are through its file number. For example, if in a program, 3 files are opened with the statement

```
10 FILES A,B,C
```

then file A will be referred to as file #1, file B as #2, and file C as #3. If the statement

```
ASSIGN "D",1,R
```

appears later in the program, file D becomes file #1, effectively closing file A. (Of course, files A,B,C, and D must have been previously created in your library.) Refer to Section XI for a complete description of the FILES and ASSIGN statements.

Opening a file also sets its file pointer to the first word of the first record of the file. There is one file pointer for each opened file in a program and it points to the place in the file where the next data item will be printed or to the location of the next data item to be read from the file.

All files in a program are automatically closed upon program termination if they have not already been closed during program execution with an ASSIGN statement.

ACCESSING BASIC FORMATTED FILES

There are two ways to store and access data in BASIC formatted files: serial access and direct access. In *serial access* items are entered in (or read from) the file sequentially. Each new item is printed immediately following the last item in the file as shown in figure 5-4.

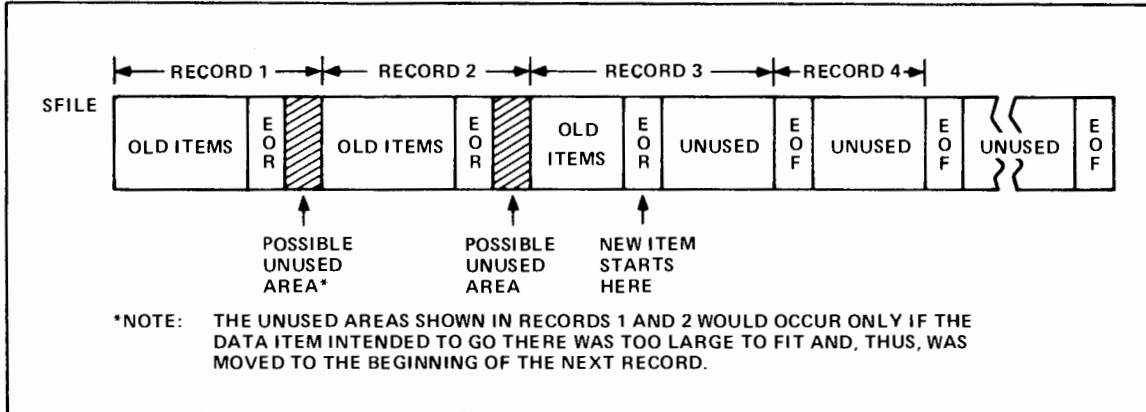


Figure 5-4. Serial File Organization

When reading an item from the file, all preceding items in that file must be read first.

The other method is *direct access*. In this case a particular record in the file can be specified so the record can be read or printed independently of the rest of the file. Figure 5-5 illustrates organization of a file accessed directly.

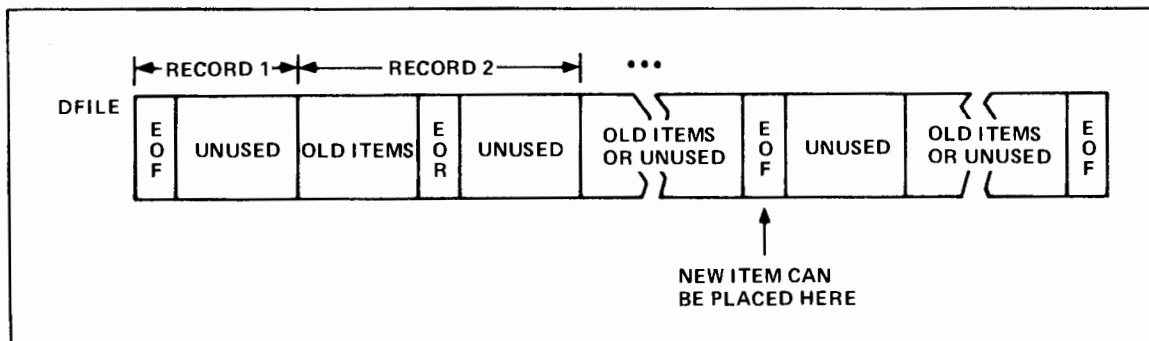


Figure 5-5. Direct File Organization

SERIAL FILE ACCESS.

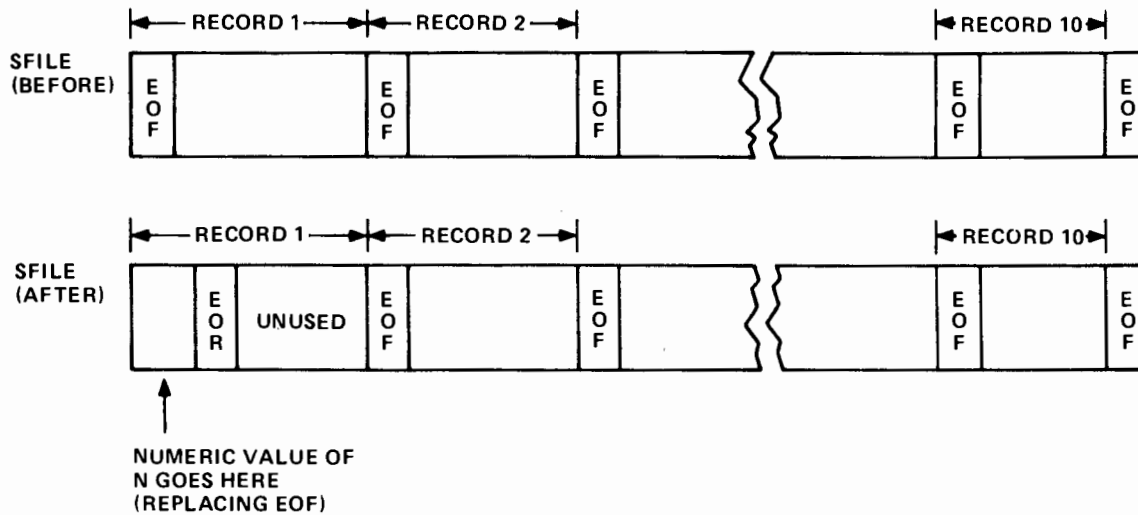
Printing Serially to a File. Items are entered serially in a BASIC formatted file using the file PRINT statement. In the following program, statement 30 is a file PRINT statement that causes the numeric value of N to be entered at the beginning of SFIDE.

CRE-SFILE, 10 ← reserves 10 consecutive records on disc for file named SFILE

```

10 FILES SFILE
20 N=1975
30 PRINT #1;N
40 END

```

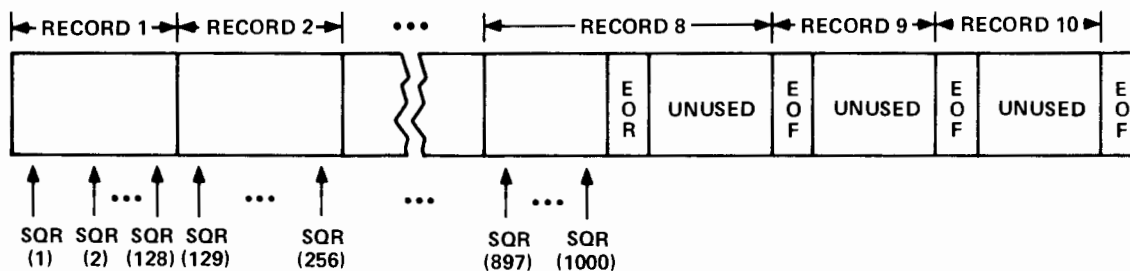


If 1000 values of N were generated and printed to SFILE, the items would be located one right after the other starting at the beginning of record 1.

```

10 FILES SFILE
15 FOR I=1 TO 1000
20 N=SQR(I)
30 PRINT #1;N
35 NEXT I
40 END

```



Note that in this example the data exactly fills the 256 words of each record. This leaves no room for the end-of-record mark in records 1 through 7. The system nevertheless knows where one record ends and another begins.

The preceding examples print numeric items to a file. String items may also be printed. For example, the serial file PRINT statement

```
PRINT #1;N$
```

is used to write a *string* N\$ to file number 1.

Files

When an item does not fit entirely at the end of a record it is automatically entered at the beginning of the next record. If a program tries to enter data when there is no more room in the file, the following message is sent to your terminal:

```
END-OF-FILE/END OF RECORD IN LINE n (where n is the line number in your program)
```

Unless the IF END statement has been specified, your program terminates.

The IF END statement can be used to direct the program to another statement, instead of terminating, when the pointer has reached an end-of-file mark. For example, if the statement

```
12 IF END #1 THEN 40
```

is inserted in the above program, then when the end-of-file mark is read, the program automatically goes to statement 40. No error message is generated. Note that the IF END statement need not be executed each time you check for an end-of-file. Once executed, the system branches to the statement number specified in the IF END statement whenever an end-of-file is encountered for the particular file number.

End-of-file marks can be programmatically written within a file with the statement

```
PRINT #1;END
```

An end-of-file mark occupies 1 word in a file except when it is at the very end of that file. Subsequent examples of reading from a file demonstrate that it is often helpful to write an end-of-file mark after the last data item in the file.

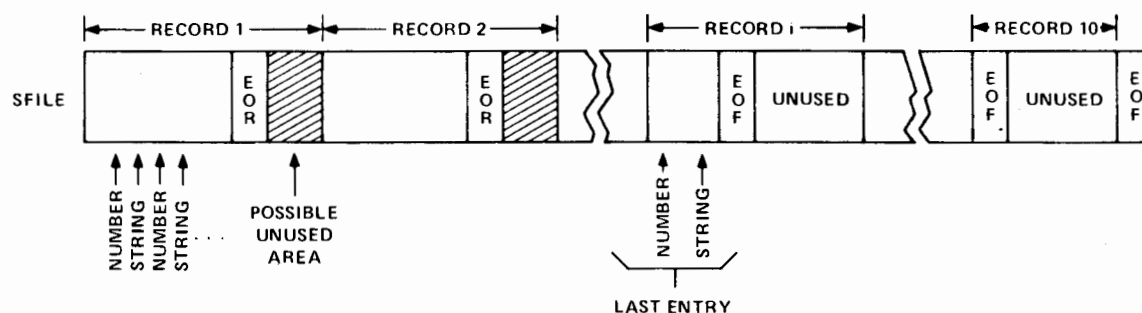
The first serial file PRINT statement in the following program (statement 40) prints a number (N) and a string (N\$) on the file SFILE. The program loops past statement 40 100 times, printing 100 entries (N, N\$) sequentially on the file. Statement 50 writes an EOF after the last entry in the file.

```
10 FILES SFILE
15 DIM N$(40)
20 FOR I=1 TO 100
30 INPUT N,N$
40 PRINT #1;N,N$
45 NEXT I
50 PRINT #1; END
60 END
```

When you run this program, statement 30 prompts you for a number and a string. You might, for example, enter an ID number for N and an employee's name for N\$.

```
RUN
```

```
?1032,"JACK FROST"
?739,"JANE EVANS"
?
```



Note that because the string that you enter each time for N\$ is variable in length, the number of records filled by this program will also vary.

Reading Serially From A File. The file READ statement is used to read information from a file. If SFILE contains the ID numbers and names of twenty employees followed by an EOF mark then statement 30 in the following program repeatedly reads a number and a name from SFILE and prints it at your terminal. When the program reads the EOF mark it branches to statement 60.

```

10 FILES SFILE
20 DIM N$(40)
25 IF END #1 THEN 60
30 READ #1;N,N$ ← reads number and name
40 PRINT N,N$ ← print to terminal
50 GOTO 30 ← go back to READ statement
60 END

```

RUN

```

1032          JACK FROST
739           JANET EVANS
250          ORVILLE PRESCOTT
330          MARY BELL
4897         LARRY SOMMERS
510          MARION EVERT
6502         GARY WIND
421          SANDY RATHER
25           GEORGE DEL CARLO
.
.
7691         RALPH PELLEGRINI

```

DONE

Changing Items In A File. Items in a file can be changed with the file UPDATE statement. For example, assume that a school has created an inventory file, INVNT, to keep track of the number of copies of each textbook in stock. Assume also that the name of each textbook (T\$) and the corresponding number of copies available (C) have already been entered in the file and an EOF mark placed after the last entry. The following program updates the file to reflect changes in the number of copies lost or added during the year.

This program reads a title from the file INVNT and prints the title at your terminal. Then it prompts you for the number of copies of that textbook. Statement 60 replaces the value of C with the value you have just entered at your terminal. The program terminates when you enter -1 as a value for C.

```

10 FILES INVNT ←———— sets file pointer at beginning
20 DIM T$(50)      of the file
30 IF END #1 THEN 80
40 READ #1;T$ ←———— read a title
50 PRINT T$ ←———— print title at terminal
55 INPUT C ←———— enter number of copies available
56 IF C=-1 THEN 80 (or -1 if you want program to end)
60 UPDATE #1;C ←———— update file
70 GOTO 40
80 END
RUN

```

```

THE YEARLING
?6
SWISS FAMILY ROBINSON
?2
THE RAILROAD FAMILY
?5
MISS BIANCA
?-1

DONE

```

The file UPDATE statement can also be used to change *strings* stored in a BASIC formatted file. The length of the new string should be the same as or shorter than the string originally written on the file. Items printed to a BASIC formatted file are written one right after another leaving no space between items. If you try to replace a string of 15 characters with a new string of 20 characters, the last five characters of the new string are truncated. You can, however, replace a 15 character string with a 10 character string since the extra characters are replaced automatically with blanks.

Recall that there is one file pointer for each file in a program and that opening a file sets its file pointer at the beginning of the first record of the file. Serial file PRINT, READ, and UPDATE statements move the file pointer through the file as each item is referenced. The file pointer is then positioned at the end of that item and it remains there until it is moved by another statement or until the program ends.

Positioning To Items In A File. The file ADVANCE statement can be used to move the file pointer past a specified number of items in the file. For instance, suppose you do not want to update the first twenty textbooks in the INVNT file. As the program now stands you would still have to read every title in the file and enter a value for the number of copies available, even if that value is unchanged. By adding the statement

```
35 ADVANCE #1;40,Z
```

to the program, the file pointer is moved past the first 40 items in the file. The updating cycle begins at the 41st item in the file. The variable *Z* is a *return variable*. Though it is a required parameter of the file ADVANCE statement and hence must be included, it serves no function in the program below. A subsequent example demonstrates the use of this *return variable*.

```

10  FILES INVNT
20  DIM T$(50)
30  IF  END #1 THEN 80
35  ADVANCE #1;40,Z
40  READ #1;T$
50  PRINT T$
55  INPUT C
60  UPDATE #1;C
70  GOTO 40
80  END

```

RUN

Statement 30 terminates the program when an EOF mark is read. Statement 35 advances the file pointer past the first 20 pairs of data items and sets *Z* to the number of items yet to be skipped, in this case, zero.

The ADVANCE statement return variable can be used to control the flow of a program when an EOF mark is read. If the ADVANCE statement encounters an EOF mark before the specified number of data items have been skipped, the return variable is set to the number of items yet to be skipped. If the statement is executed successfully, the return variable is set to 0. As an example, suppose that the school in the previous example had a fire in the supply room during the summer and every textbook was lost. The following program could be used to bring the INVNT file up-to-date, setting *C* (the number of copies available) to 0 for every textbook in the file. The statement

```
30 ADVANCE #1;1,Z
```

causes the program to skip one item, in this case *T\$*, the title of the text. You can then set *C* to 0 using the statement

```
60 UPDATE #1;0
```

The program repeatedly skips *T\$* and replaces the old value of *C* with a 0. The return variable *Z* is set to zero until the EOF mark is read and it is set to 1. When *Z* is no longer zero, the program branches to statement 70 and end.

```

10  FILES INVNT
20  DIM T$(50)
30  ADVANCE #1;1,Z ←———— skips T$, sets return variable;
40  IF Z <> 0 THEN 70      if no EOF is found, Z = 0
50  UPDATE #1;0 ←———— replace old value of C with 0
60  GOTO 30
70  END

```

A special updating technique takes advantage of the fact that the same file may be assigned to more than one file number. Since a buffer is created in memory for each file number, this provides a file with more than one buffer area into which data is read or from which data is printed.

For example, the following statement equates file AFILE to file number 1 and 2:

```
10 FILES AFILE, AFILE
```

The feature can be used in sorts where you want to maintain the order of the file in one buffer while re-arranging it in another. Another use is for certain updates. Suppose you want to remove, for instance, every third item in a serial file. Assuming file AFILE has been assigned to file number 1 and 2:

```
15 DIM A$(10), B$(10), C$(10)
20 READ #1; A$, B$, C$
30 PRINT #2; A$, B$
40 GOTO 20
50 END
```

When this technique, known as double buffering, is used for direct file access, it is important to read a record following each PRINT statement. This insures that each record to which new information is printed is actually written to the disc.

DIRECT FILE ACCESS. Printing to a file using serial access enters each item one directly after another, so that space on the disc is used efficiently. When there is a lot of data in the file, however, serial access can be slow. To access a specific data item in a file using serial access, all data located in front of that item must be accessed. *Direct access* enables you to go directly to a particular record in the file. The statements and functions used for direct file access differ from those used for serial file access only in an additional parameter, the *record number*, specified in the file READ and PRINT statements. For example,

Serial Access	Direct Access
100 READ #1;A,B,C	100 READ #1,5;A,B,C
110 PRINT #2;A,B,C	110 PRINT #2,2;A,B,C

These serial access statements read three numeric values from file number 1 starting wherever the number 1 file pointer is located, then writes them on file number 2 at the location of the number 2 file pointer. The direct access statements read three numeric values from the fifth record in file number 1 and write them at the beginning of the second record in file number 2.

By planning ahead you can arrange the items in a file so that you can move the file pointer directly to the item you want without reading through all preceding items. The following examples demonstrate the use of direct file access for keeping track of the number of textbooks in stock.

Printing Directly to a File. Instead of storing each title (T\$) and the corresponding number of copies (C) one right after another in the file, suppose you would like to group the texts in five categories: science, mathematics, English, languages, and history and you want to store them in separate areas within the same file. The first step, as usual, is to determine how long the file is going to be and to reserve space on the disc using the CREATE command. For simplicity, plan to enter a different text in each record. Allowing for a maximum of 10 texts in each of the 5 categories you should create a file 50 records long:

```
CRE-TEXTS,50
```

The following program is used to initially enter the items in the file. The program asks you which record number you want to write to and prompts you to enter a title and a value for C (number of copies). When you enter a 0 for the record number, the program ends.

```

10 FILES TEXTS
20 DIM T$(50)
25 PRINT "WHICH RECORD?"
30 INPUT R
32 IF R=0 THEN 80 ← enter 0 for R if you're finished
35 PRINT "WHAT'S THE TITLE?"
40 INPUT T$
45 PRINT "HOW MANY COPIES?"
50 INPUT N
60 PRINT #1,R;T$,N ← prints values of T$ and N to
70 GOTO 25          TEXTS file at record R
80 END
RUN

```

```

WHICH RECORD?
?1
WHAT'S THE TITLE?
?"SCIENCE 1"
HOW MANY COPIES?
?30
WHICH RECORD?
?2
WHAT'S THE TITLE?
?"SCIENCE 2"
HOW MANY COPIES?
?20
WHICH RECORD?
?11
WHAT'S THE TITLE?
?"MATH 1"
HOW MANY COPIES?
?30
WHICH RECORD?
?13
WHAT'S THE TITLE?
?"MATH 3"
HOW MANY COPIES?
?10
WHICH RECORD?
?0
DONE

```

Files

Reading Directly from a File. The next program uses a direct file READ statement to tell you how many copies of a book you have. It asks you to enter the record number and then prints out the contents of that record. When you enter a 0 for the record number, the program ends.

```
10 FILES TEXTS
20 DIM T$(50)
25 PRINT "WHICH RECORD?"
30 INPUT R
32 IF R=0 THEN 70
40 READ #1,R;T$,N
50 PRINT T$,N ← prints at terminal
60 GOTO 25 ← go back and enter another
70 END record number
RUN
```

```
WHICH RECORD?
?2
SCIENCE 2      20
WHICH RECORD?
?13
MATH 3        10
WHICH RECORD?
?0

DONE
```

If you type the number of a record that is empty, the message

```
END-OF-FILE/END OF RECORD IN LINE 40
```

is returned and the program terminates.

When reading either serially or directly from files, it is useful to be familiar with the TYP function. When the TYP function is used to test data in a file it returns a value indicating whether the next item in the file is a number, a string, an end-of-file mark, or an end-of-record mark. The function TYP(N) tests the next item in the file N. If that item is a number, TYP(N) returns a value of 1. If that item is a string, a value of 2 is returned. An EOF mark returns a value of 3. To test for EOR marks, the value of N must be negative. The return value for an EOR mark is 4.

For example, suppose you would like to generate a list of the textbooks available in the TEXTS file. One way to do this is to read each item, test it to see if it is a string, and if so, print it. If the item is a string, a value of 2 is returned to the TYP function.

```

10 FILES TEXTS
20 DIM T$(50)
30 FOR R=1 TO 50
40 READ #1,R
50 IF TYP(1) <> 2 THEN 70 ← when you create the file EOFs are automatical-
                           ly placed in the first word of each record. If
                           there's nothing there, skip over READ.
55 READ #1;T$ ← get the string from the file
60 PRINT T$
70 NEXT R ← go to the next record
80 END
RUN

```

```

SCIENCE 1
SCIENCE 2
SCIENCE 3
ELEMENTARY ARITHMETIC
MATH 1
MATH 2
MATH 3
MATH7
BIOLOGY 2

DONE

```

Refer to Section XI for a more detailed explanation of the TYP function.

Updating a File Directly. Two other useful 2000 BASIC functions are the REC function and the ITM function. The REC function returns the current record number being accessed in the file. The ITM function returns the number of data items between the beginning of the currently accessed record and the position of the file pointer. For example, the following program is used to change the value of C (number of copies) in the INVNT file. After a title (T\$) is read the file pointer is positioned at the corresponding value of C. Statements 50 and 60 record what this position is by setting R to the record number and I to the item number. The value of C is then read and printed. At this point the file pointer has moved beyond the value of C in the file. In order to update that value of C the file pointer must be repositioned. The file pointer cannot be moved backward within a file but it can be positioned to a preceding record with a file READ statement. Statement 110 sets the file pointer to the beginning of record R; statement 120 advances the pointer to item I, and statement 130 updates the value of C. The program loops until a -1 is entered for the number of copies (C) or until the pointer reads an end-of-file mark.

```

10 FILES INVNT
20 DIM T$(50)
30 IF END #1 THEN 150
35 PRINT "ENTER NEW NUMBER OR ELSE -1 TO TERMINATE"
40 READ #1;T$
50 R=REC(1)
60 I=ITM(1)
70 READ #1;C
80 PRINT "NO. OF COPIES OF "T$" IS NOW "C
90 INPUT C
100 IF C=-1 THEN 150
110 READ #1,R
120 ADVANCE #1;I,X
130 UPDATE #1;C
140 GOTO 40
150 END
RUN

```

```

ENTER NEW NUMBER OR ELSE -1 TO TERMINATE
NO. OF COPIES OF THE YEARLING IS NOW 6
?5
NO. OF COPIES OF SWISS FAMILY ROBINSON IS NOW 2
?2
NO. OF COPIES OF THE RAILROAD FAMILY IS NOW 5
?-1

```

DONE

Table 5-1. Summary of File Statements and Functions

STATEMENT OR FUNCTION	USE
PRINT #	writes items to a file; serial and direct
READ #	reads items from a file; serial and direct
UPDATE #	changes items in a file; serial only, cannot specify record number
ADVANCE #	advances the file pointer; serial only, cannot specify record number
IF END # THEN	causes program to branch when an EOF mark is read; serial and direct (direct access takes EOF exit if you try to read or print past the end of a record)
ITM function	returns number of data items between the beginning of the currently accessed record and the position of the file pointer
REC function	returns the current record number being accessed in the specified file
TYP function	tests type of next item in file — number, string, EOF, or EOR

ASCII FILES

ASCII files provide you with programmable access to the following devices: magnetic tape units, paper tape readers, paper tape punches, card readers, card reader/punch/interpreters, line printers, and link terminals. ASCII disc files complement BASIC formatted files by providing additional means to store and access information on the HP 2000 System discs. In addition, ASCII files provide a means to access remote computer systems. Whether you have the capability of accessing ASCII files is determined by your system master. When your account is created, it may include the ability to reference some or all of the system's peripheral devices.

Every ASCII file consists of a sequence of records, each of which contains exactly one string. The last record of an ASCII file is followed by an EOF mark.

CREATING AND PURGING ASCII FILES

ASCII files are equated to a device or to an area on the disc with the FILE command. Each FILE command includes specification of a name (1 to 6 letters or digits) and a device type. The record length of an ASCII device file is an optional parameter. The maximum record length for each device, specified in *words*, is determined by the longest string item that can be read or printed to that device. Recall that 1 word contains 2 string characters. Thus an 80-column card has a maximum record length of 40 words.

The DEVICE command displays a list of devices available to you. A typical device report is shown.

DEVICE DESIGNATOR	MAXIMUM RECORD SIZE	STATUS
CR0	40	
LP0	66	BUSY
LP1	66	
PPO	64	
JM0	60	
JI0	36	
JPO	41	
JP1	41	
JP2	41	
JL0	67	
JL1	67	
JL2	67	
JT0	40	
JT1	40	
JT2	40	
JT3	40	
MT0	256	N/A
PRO	64	

The items in the first column are mnemonics specified by the system operator that represent the various devices available. For instance, CR0 is card reader number 0; LP0 and LP1 are line printers 0 and 1. The mnemonics that begin with the letter J refer to job function designators. An ASCII file created as a job function designator can be used to transfer jobs between an HP 2000 system and an IBM or CDC host system or to transfer data between two HP 2000 systems. A job inquiry file is used to send messages to the host, a job transmitter to send jobs. The job message file receives messages from the host; job lister and job punch files receive output from the host. When using these files, you print to a file equated to a JI or JT device and read from a file equated to a JL, JP or JM device. Refer to Section IX for an explanation of these devices in data communications.

The items in the second column, the maximum record size column, show how many words each record may contain for each device. This maximum record size was chosen by the system operator when your system was configured. The numbers shown in the listing are typical.

The status column shows that LP0 is busy. If you were to request that specific device, your program would terminate with the message:

NO ACCESS AT THIS TIME

An N/A entry in the STATUS column indicates that the system operator has removed that device from the system or has given another user exclusive access to that device. (Refer to the HP 2000 Computer System Operator's Manual for a complete discussion of system operator capabilities.)

Disc resident ASCII files, like BASIC formatted files, are available to all users. In addition to the file name, device type (DS for disc), and optional maximum record length, the FILE command for ASCII disc files must specify the number of *blocks*. ASCII disc files pack as many records as are possible in each 256 word block of disc space. The default value for ASCII disc file maximum record length is 63 words. Four 63 word records would be packed in 1 block of disc space.

Note that ASCII disc files differ from BASIC formatted files in these respects:

- A BASIC formatted file always stores one record per 256-word block; an ASCII disc file can store as many records as will fit in a 256-word block.
- The default record size for BASIC formatted files is 256 words; for ASCII files, the default maximum record size is 63 words.
- Record size for BASIC formatted files may be specified as a value between 64 and 256; maximum record size for ASCII files may be specified as a value between 1 and the maximum size allowed for the device (255 for disc).
- File length for BASIC formatted files is specified as the number of 256-word records; for ASCII disc files as the number of 256-word blocks.
- BASIC formatted files may be created with either the CREATE command or CREATE statement; all ASCII files are created with the FILE command.

Note also that if an ASCII file contains BASIC language statements, these can be loaded into your work space as a program using the LOAD command (refer to Section X). This allows you to store a program on a disc file, rather than in off-line storage, and subsequently bring it into your work space for execution. This cannot be done with BASIC formatted files.

Examples:

<code>FILE-ASC1, CR0</code>	ASC1 is an ASCII file equated to the card reader CR0.
<code>FILE-ASC2, DS, 50</code>	ASC2 is an ASCII disc file with 50 blocks and a maximum record length of 63 words.
<code>FILE-ASC3, LP1</code>	ASC3 is equated to line printer 1. The record length is the default established at system configuration (normally 66 words for a line printer).
<code>FILE-ASC4, MT, 80</code>	ASC4 is equated to any magnetic tape unit. The maximum record length is 80 words.

Note: These ASCII files will be used in later examples in this section.

The device designators in these four FILE commands are CR0, DS, LP1, and MT. CR0 and LP1 are *specific* device designators that name a specific non-sharable device or job function designator. DS and MT are *general* device designators that specify non-sharable devices and job function designators by *type*. If you specify a general device for ASCII file input/output, the system assigns your program to any available device of that type on the system.

ASCII files can be purged with the PURGE command.

OPENING ASCII FILES

Once an ASCII file has been created, it can be opened for use in a program by either a FILES statement or an ASSIGN statement.

```
10 FILES ASC1,ASC2,ASC3
```

Opening a file causes the file to be associated with an integer file number from 1 to 16, according to its position in the FILES statement. In the example above, ASC1 becomes file #1, ASC2 is file #2, and ASC3 is file #3. Once a file is opened, all references to it are through its file number. The ASSIGN statement can be used in a program to re-assign file numbers. For instance, the following statement opens the file ASC4 and associates ASC4 with file #2, closing the file ASC2:

```
20 ASSIGN "ASC4",2,R
```

PRINTING TO ASCII FILES

Data is written to an ASCII file using the PRINT # statement. Only serial access (not direct) is available for ASCII files.

A very common use of ASCII files is to have data printed on a line printer. The following program opens the ASCII file associated with the line printer, LP1, and prints a table of 10 integers, their squares, cubes, and square roots on the file:

```
10 FILES ASC3
20 FOR I=1 TO 10
30 PRINT #1;I,I ** 2,I ** 3,SQR(I)
40 NEXT I
50 END
```

Data is printed to an ASCII file using the same format rules that are used for printing at your terminal. In this case, the output is printed on the lineprinter as shown below.

RUN

1	1	1	1
2	4	8	1.41421
3	9	27	1.73205
4	16	64	2
5	25	125	2.23607
6	36	216	2.44949
7	49	343	2.64575
8	64	512	2.82843
9	81	729	3
10	100	1000	3.16228

Numeric and string data items separated by commas are printed one per field (where a field is 15 positions wide); they are printed closely packed if separated with semicolons. The record length of an ASCII file determines the number of print fields per record. When printing string data, a print that generates a line longer than the record size of an ASCII file results in truncation of the line by dropping excess characters from the right end of the line. When printing numeric fields, excess fields are printed on following lines. (Refer to the PRINT statement in Section XI for a complete description of output format.)

Printing to ASCII file #0 directs the output to your terminal. Thus the following two statements are equivalent:

```
35 PRINT I, I**2, I**3, SQR(I)
```

```
35 PRINT #0; I, I**2, I**3, SQR(I)
```

ASCII file #0 is useful for programmatic assignment of output devices, e.g.

```
6900  FILES ASC3
7000  PRINT "DO YOU WANT OUTPUT ON THE LINEPRINTER?"
7010  INPUT A$
7020  IF A$="Y" THEN 7050
7030  F=0
7040  GOTO 7060
7050  F=1
7060  PRINT #F; I, I ** 2, I ** 3, SQR(I)
9000  END
```

Recall that the file ASC3 was previously associated with LP1 through the command:

```
FILE-ASC3,LP1
```

The PRINT #; USING statement and the MAT PRINT #; USING statement provide more control over the format of your output. For example, the statements

```
30 PRINT #1; USING 35; I, I**2, I**3, SQR(I)
```

```
35 IMAGE 10D, 10D, 10D, 10D.2D
```

are used in the following program to specify the output format for the table of integers, squares, cubes, and square roots.

```
10 FILES ASC3
20 FOR I=1 TO 10
30 PRINT #1; USING 35; I, I ** 2, I ** 3, SQR(I)
35 IMAGE 10D, 10D, 10D, 10D.2D
40 NEXT I
50 END
```

RUN

1	1	1	1.00
2	4	8	1.41
3	9	27	1.73
4	16	64	2.00
5	25	125	2.24
6	36	216	2.45
7	49	343	2.65
8	64	512	2.83
9	81	729	3.00
10	100	1000	3.16

DONE

The rules for formatting output with the PRINT #; USING and MAT PRINT #; USING statements are the same as explained in Section XI for the PRINT USING and MAT PRINT USING statements.

Note that you cannot use the PRINT # USING statement with a BASIC formatted file; also the PRINT #0; USING statement is identical to the PRINT USING statement.

USING CTL FUNCTION WITH ASCII FILES. The control function (CTL) permits additional user control of ASCII devices. It is used in ASCII file PRINT statements to cause the line printer to skip lines or pages or to suppress paper advances; it can advance or rewind magnetic tapes; it can be used to reset the pointer in an ASCII disc file; and it can control parity and record separators when punching a paper tape. (Refer to the CTL function and ASCII File Print in Section XI for a complete description of the CTL capabilities.) For example, the function CTL(24) sent to a magnetic tape unit will rewind the tape currently mounted on the unit. If CTL(24) is sent to an ASCII disc file, it resets the file pointer to the beginning of the file. CTL(24) is not defined for any other non-sharable device and will be ignored if it is sent to one. (A complete list of CTL arguments and their use is contained in Section XI.)

The following program reads a deck of cards printing 25 cards on each line printer page. The CONTROL function CTL(1) advances the paper to the top of the next page.

```

10  FILES ASC1,ASC3
20  DIM A$(80)
30  IF END #1 THEN 100
40  FOR R=1 TO 25
50  LINPUT #1;A$
60  PRINT #2;A$
70  NEXT R
80  PRINT #2;CTL(1)
90  GOTO 40
100 END

```

USING *OUT=filename* WITH ASCII FILES. ASCII file devices can also be accessed using the *OUT= file name* forms of the RUN, EXECUTE, LIST, and PUNCH commands (refer to Section XI). For instance, to obtain a listing on the line printer, list your program using the following command:

```
LIST*OUT=ASC3*
```

The file must be an ASCII file supporting output (e.g. line printer, paper tape punch, magnetic tape, disc), defined using the FILE command. For example:

```
FILE-ASC3,LPO
```



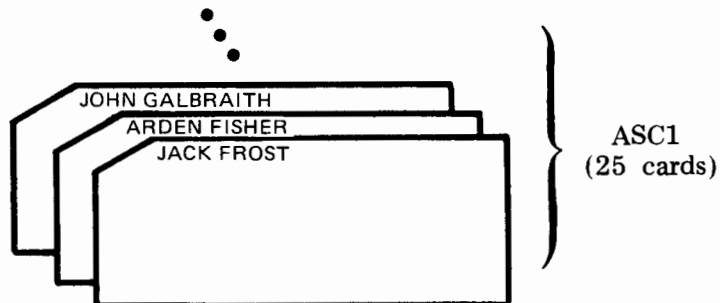
READING FROM AN ASCII FILE

To read from an ASCII file use either a file READ or a file LINPUT statement. For example, the following program reads names from a stack of 25 cards and prints the names at your terminal. (File ASC1 was previously associated with the card reader using the FILE command.)

```
10  FILES ASC1
15  DIM N$(40)
20  FOR I=1 TO 25
25  READ #1;N$
30  PRINT N$
35  NEXT I
40  END
```

```
RUN
```

```
JACK FROST
ARDEN FISHER
JOHN GALBRAITH
MARTIN DYER
RACHEL MCKAY
JOHN THOMPSON
GRANT NELSON
CORNELIA HOCHBERG
DONALD MARTIN
ANITA KANE
```



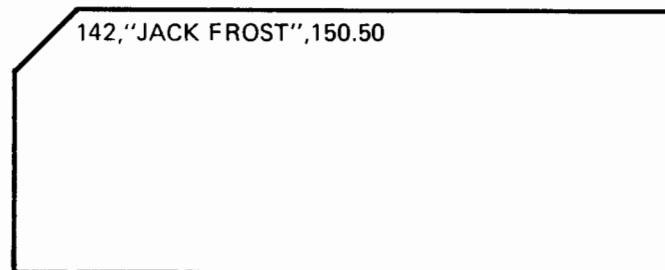
Files

If the data on the cards also included employee ID numbers and salaries, the file READ statement would be:

```
25 READ #1;X,N$,S  
RUN
```

142	JACK FROST	150.50
2156	ARDEN FISHER	300.75
113	JOHN GALBRAITH	975.00
1289	MARTIN DYER	115.25
130	RACHEL MCKAY	375.75
267	JOHN THOMPSON	190.00
187	GRANT NELSON	400.00
265	CONNIE HOCHBERG	350.00
119	DONALD MARTIN	575.50
172	ANITA KANE	315.00

ASCII file read operations require commas between data items exactly as if the data were input from your terminal. The data on each card for the above program would have to be arranged as shown:



142,"JACK FROST",150.50

Note that strings must be quoted unless they are the only or last item in the list. Also note that ASCII file PRINT statements do not automatically print commas between the data items. When printing data to an ASCII file that you want to read later using a file READ statement, you must include a comma as a *literal string* (“,”) between items in the write list. For example, suppose you were writing data to an ASCII magnetic tape file. The statement

```
20 PRINT #1;142, "JACK FROST", 150.50
```

would print the following string on the magnetic tape file:

```
142    JACK FROST    150.50
```

The commas between items in the PRINT statement are not written to the tape; as with any PRINT statement, they simply cause the items to be separated into 15-column fields.

If you later wanted to read that data using a file READ statement, i.e.

```
120 READ #1; I,N$,S
```

the message

```
DATA OF WRONG TYPE IN 120
```

would be returned to you. ASCII file read operations require commas between items. To meet this requirement the PRINT # statement would have to be:

```
20 PRINT #1;142, ",", '34 "JACK FROST" '34, ",", 150.50
```

(34 is the decimal equivalent of the ASCII quotation mark)

This would print the following string, *including* commas, on the magnetic tape:

```
142      ,      "JACK FROST"      ,      150.50
```

The file READ statement

```
120 READ #1; I,N$,S
```

can now execute as intended.

The file LINPUT statement accepts an entire line of string data and assigns it to a destination string. All characters entered, including commas, quote marks, and leading and trailing blanks are assigned to the string.

File #0, your terminal, may also be used with the file READ and file LINPUT statements. The statement

```
120 READ #0;I,N$,S
```

causes your program to read an ID number, name, and salary from your terminal. It is equivalent to the statement

```
120 INPUT I,N$,S
```

The statement

```
120 LINPUT #0;N$
```

is equivalent to the statement

```
120 LINPUT N$
```


Table 5-2 summarizes the major differences between BASIC formatted files and ASCII files:

Table 5-2. BASIC Formatted Versus ASCII File Usage

BASIC FORMATTED FILES	ASCII FILES
Always present on-line	Not always present on-line
Data can only be created or retrieved on-line through programs	Data can also be created and retrieved off-line fairly easily
Numbers and strings can be intermixed without ambiguity	Numbers and strings may not be distinguishable outside of the context of an accessing program
Files can be accessed serially or directly	With few exceptions, files can only be accessed serially
Individual records can be altered without affecting other records	The ability to selectively alter portions of a file is restricted or non-existent, depending on the device being used
Always on disc	Can be on disc, magnetic tape, line printer, card reader, paper tape punch, paper tape reader, card reader/punch interpreter, or link terminal.
Read and write access	Except for ASCII disc files, magnetic tape and link terminal files, a given file is read-only or write-only.
Multiple access possible	No multiple access possible (hence the term "non-sharable device")
Programs cannot be stored for later execution	Programs stored on ASCII files can be retrieved for later execution

WHAT IS FORMATTED OUTPUT?

Formatted output gives you explicit and exact control over the format of your program output.

- Numbers can be printed in three different representations: integer, fixed point, and floating point.
- The exact position of plus and minus signs can be specified.
- String values can be printed in specified fields and literal strings and blanks can be inserted wherever needed.
- You can have full control over carriage returns and line feeds.
- Arbitrarily long lines can be printed without the carriage returns and line feeds that PRINT and MAT PRINT statements normally provide.

HOW DO YOU INDICATE FORMATTED OUTPUT?

The PRINT USING and MAT PRINT USING statements provide formatted output at your terminal and the file statements PRINT # USING and MAT PRINT # USING send formatted output to ASCII files.

There are two key elements in formatted output, the list of items to be printed (the using list) and the format in which these items are to be printed (the format string).

These elements are illustrated in the following statement:

```
40 PRINT USING "SDD.DD"; A,B,C
```

format string *using list*

The *format string* is either a quoted string or a string variable when specified in a PRINT USING statement; when it is part of an IMAGE statement it is an unquoted string of characters.

The *using list* in this case consists of simple numeric variables; it could also consist of numeric expressions, string variables, array elements, or the print functions LIN, SPA, TAB, and CTL.

When the statement is executed, it prints the contents of A, B, and C according to the format "SDD.DD". This format specifies that a sign (+ or -) precedes two decimal digits followed by a decimal point and two more decimal digits.

Formatted Output

Of course, what is printed depends on the contents of A, B, and C. To continue the illustration, set values in these variables and run the example:

```
10 LET A=10.35
20 LET B=-.304
30 LET C=1.0359
40 PRINT USING "SDD.DD";A,B,C
50 END
```

When executed, the values in A, B, and C are formatted according to the format string SDD.DD as follows:

```
  +10.35  -0.30  +1.04
  └───┬──┘ └──┬──┘ └──┬──┘
      A      B      C
```

Note that the numbers are rounded, and the sign is shifted to immediately precede the first significant digit. If there are more significant digits to the left of the decimal point than there are positions (D's) in the format, the value is not formatted but is printed as a floating-point number on a separate line.

The statement number of an IMAGE statement may be specified in a PRINT USING statement instead of the format string. The IMAGE statement contains a format string without quotes. For example:

```
40 PRINT USING 45;A,B,C
45 IMAGE SDD.DD
```

These two statements perform exactly the same function as statement 40 in the preceding example.

Similarly, a string variable could contain the format. For instance,

```
30 FS="SDD.DD"
40 PRINT USING FS;A,B,C
```

USING LIST

The using list specifies the items to be printed. These items include numeric expressions and constants, string variables (but not string constants), and print functions. The print functions include:

TAB(x)	Tabulate to column x before printing next item.
SPA(x)	Skip x spaces before printing next item.
LIN(x)	Skip x lines before printing next item; if x is negative, carriage return is suppressed, if zero, only a carriage return is generated.
CTL(x)	Perform control function on ASCII file specified in the print statement; function depends on value of x; used only for ASCII files.

Spaces in a format string can be used to perform the same function as SPA(x).

Each item in the using list is separated from the next by a comma. Such commas are separators only and do not perform the print formatting functions they perform in a PRINT or MAT PRINT statement.

FORMAT STRING

The examples so far have illustrated a simple format for representing decimal numbers. A format string may contain:

- Numeric formatting characters (D S . E X)
- String formatting characters (A X)
- Repetition factors (integer from 1-255)
- Delimiters (, /)
- Carriage control characters (+ - #)
- String constants

NUMERIC FORMATTING CHARACTERS. The characters used to define numeric formatting are:

- D One D for each digit to be printed.
- S Once only, to print the sign of the number.
- . Once only, in the position where a decimal point is to be printed.
- E Once only, following other characters, to print the exponent of the number.
- X One X for every blank space.

Of these, the decimal point (.) and the E are actually printed; D, S, and X reserve positions for a digit, a sign, or a space respectively.

STRING FORMATTING CHARACTERS. Only two string formatting characters are used:

- A One A for each character to be printed.
- X One X for each blank space.

The X in numeric formats and in string formats performs the same function. Otherwise, string and numeric formatting characters cannot be combined. String formats are used to format string output, numeric formats to format numeric output.

REPETITION FACTORS. Rather than repeating the characters D, X, or A, you can precede these characters with an integer to indicate repetition. For example, the format string "DDDD" is the same as "4D" and "2D4X2D" is the same as "DDXXXXDD."

In addition, a group of format characters can be repeated if they are enclosed in parentheses and preceded by an integer. Refer to the PRINT USING statement description in Section XI for a discussion of groups.

Formatted Output

Examples of Numeric and String Formats

Integer output:

<u>Format Specification</u>	<u>Value</u>	<u>Format of Output</u>
4D	1234	1234
S4D	1234	+1234
4DS	1234	1234+
5D	1234	1234
4D	1234.8	1235
DXDDD	1234	1 234
S10D	1234	+1234
DSDDD	1234	1+234
5D	-1234	-1234
4D	1234.2	1234

Fixed-point output:

<u>Format Specification</u>	<u>Value</u>	<u>Format of Output</u>
3D.4D	465.465	465.4650
4D.2D	465.465	465.47
4D.3D	-465.465	-465.465
SDD2D.D	465.465	+465.5
S2D.4D	.465	+0.4650
S.4D	.465	+ .4650
D.4D	-.465	- .4650
2D.4D	-.465	-0.4650

Floating point output:

<u>Format Specification</u>	<u>Value</u>	<u>Format of Output</u>
SDXE	4.82716 X 10 ²¹	+5 E+21
DDDD.DDE	SAME	4827.16E+18
S5DX.X5DEX	SAME	+48 . 27159E +20
SD.5DE	SAME	+4.82716E+21
S.10DE3X	SAME	+.4827159382E +22

String output:

<u>Format Specification</u>	<u>String Value</u>	<u>Format of Output</u>
6A	ABCDEF	ABCDEF
5A	ABCDEF	ABCDE
8A	ABCDEF	ABCDEF
2X6A	ABCDEF	ABCDEF
AXAXAXAXA	ABCDEF	A B C D E F

DELIMITERS. When more than one format is contained in a format string, they must be separated by either a comma or a slash (/). The comma simply acts as a delimiter; the slash serves a double function: it is both a delimiter and a carriage control character causing a carriage return and line feed.

For example, the following short program illustrates the use of more than one format, one for numeric items and one for string items:

```
5 DIM A$(10)
10 A$="ABCDEF"
20 B=213057.
30 PRINT USING "3A2X3A/S6D.DDE";A$,B
40 END
RUN
```

slash specifies carriage return & linefeed

```
ABC DEF
+21305.70E+01
```

DONE

CARRIAGE CONTROL CHARACTERS. A terminating semicolon cannot be used in a formatted print statement to suppress the carriage return/linefeed at the end of a line of print. Such control is provided by three carriage control characters that may be specified at the beginning of a format string:

- + Suppress linefeed that normally follows print line.
- Suppress carriage return that normally follows print line.
- # Suppress both carriage return and line feed.

The following program uses the characters “-” and “#”:

```
5 DIM A$(10)
10 A$="ABCDEFGH IJ"
20 B=123.45
30 PRINT USING "-,X3A,X2A";A$(1,3),A$(8)
40 PRINT USING "#,6A2X";A$
50 PRINT USING "S3D.3D";B
60 END
RUN
```

```
ABC HI
      ABCDEF +123.450
```

DONE

Formatted Output

The plus (+) causes overprinting by the next PRINT statement. This can be useful in the case of printed security codes. For example:

```
10 DIM C$(10),X$(10)
15 X$="XXXXXXXXXX"
20 C$="XK915"
30 PRINT USING 60;C$
40 PRINT USING 60;X$
50 PRINT "HHHHHHHHHH"
60 IMAGE +,10A
70 END
RUN
```

XXXXXXXXXX ← *First X's and then H's overprint the value of C\$*

DONE

This feature can also be used with PRINT # USING or MAT PRINT # USING statements to print formatted output to ASCII files associated with certain types of line printers.

Remember that the slash (/) can be used to generate a carriage return linefeed within a format string.

STRING CONSTANTS. The format string may consist simply of a string constant or it may combine a constant with other format specifications. The string constant is printed literally and if the format string contains only X's, carriage control, grouping, or replicator codes, the using list can be omitted. In an IMAGE statement, a string constant is simply quoted; in a PRINT USING statement, the numeric representation of quotation marks ('34) must surround the quoted string.

For example:

```
100 I=2.57
110 PRINT USING 120;30*I
120 IMAGE "TOTAL =",3X,6D.3D
130 END
RUN
```

TOTAL = 77.100

DONE

The following PRINT USING statement could be substituted for lines 110 and 120 with the same result:

```
110 PRINT USING '34"TOTAL ="'34",3X,6D.3D";(30*I)
```

Note the use of '34 around the string constant; the rest of the string must be quoted separately. Because this method is cumbersome, usually an IMAGE statement is used for string constants. Also, note that the following statements perform exactly the same function and the second is simpler:

```
10 PRINT USING '34"THIS IS AN EXAMPLE"'34
20 PRINT "THIS IS AN EXAMPLE" ← no using list required
```

However, in the following case, PRINT USING is simpler than using a PRINT statement:

```
10 PRINT USING 20
20 IMAGE "COLUMN1",10X,"COLUMN 2",10X,"COLUMN 3",10X,"COLUMN 4"
```

USING FORMATTED OUTPUT

The first step in producing formatted output is to decide what each item in the using list should look like when it is printed. The next step is to decide where it is to be printed. Then the format strings can be specified using carriage control characters if needed. What cannot be formatted by the format string can usually be formatted with a print function in the using list.

When the form of the output has been determined, the PRINT USING and IMAGE statements can be incorporated in your program. These statements are particularly useful for printing tables and reports. The following two programs illustrate two uses of formatted output.

Program 1:

This program prints a numeric conversion chart. It prints a value in inches from 1 through 25 followed by the equivalent value in metres, centimetres, and micrometres. Integer, fixed-point, and floating-point formats are used. The output is formatted into four columns with headings over each column.

```
100 REM PRINT THE HEADING
110 PRINT USING 120
120 IMAGE "INCHES",10X,"METRES",4X,"CENTIMETRES",4X,"MICROMETRES"
130 FOR I=1 TO 25
140 REM PRINT THE VALUE OF INCHES
150 REM USING CARRIAGE CONTROL TO SUPPRESS CARRIAGE RETURN/LINEFEED
160 PRINT USING 170;I
170 IMAGE#,2X,2D,13X
180 REM CONVERT TO OTHER UNITS
190 M=I*.0254
200 C=M*100
210 M1=M*1.E+06
220 REM PRINT VALUES IN FIXED POINT AND FLOATING POINT
230 PRINT USING ".DDD,8X,DD.2D,8X,D.3DE";M,C,M1
240 NEXT I
250 END
```


Formatted Output

When program 1 is executed, the following table is printed:

INCHES	METRES	CENTIMETRES	MICROMETRES
1	.025	2.54	2.540E+04
2	.051	5.08	5.080E+04
3	.076	7.62	7.620E+04
4	.102	10.16	1.016E+05
5	.127	12.70	1.270E+05
6	.152	15.24	1.524E+05
7	.178	17.78	1.778E+05
8	.203	20.32	2.032E+05
9	.229	22.86	2.286E+05
10	.254	25.40	2.540E+05
11	.279	27.94	2.794E+05
12	.305	30.48	3.048E+05
13	.330	33.02	3.302E+05
14	.356	35.56	3.556E+05
15	.381	38.10	3.810E+05
16	.406	40.64	4.064E+05
17	.432	43.18	4.318E+05
18	.457	45.72	4.572E+05
19	.483	48.26	4.826E+05
20	.508	50.80	5.080E+05
21	.533	53.34	5.334E+05
22	.559	55.88	5.588E+05
23	.584	58.42	5.842E+05
24	.610	60.96	6.096E+05
25	.635	63.50	6.350E+05

Program 2:

This program is a sample report generator. It first requests a school number from the terminal, then reads and prints out information about the school's teachers from a file. Note that a carriage control character is used to advantage (statement 100), slashes (/) are used (statement 200), string and fixed-point fields are used (statement 210), and an error occurs in the output for the eighth teacher (number too large for field; therefore, it is printed in E format on a separate line).

```

10  REM THIS PROGRAM GENERATES A REPORT ON TEACHERS
50  DIM A$(25),B$(19),C$(19)
60  FILES SCH1,SCH2,SCH3,SCH4,SCH5
100 IMAGE #,"ENTER SCHOOL NUMBER:"
150 IMAGE "TEACHER",13X,"SUBJECT",13X,"SALARY",4X,"ATTND."
175 IMAGE "-----",13X,"-----",13X,"-----",4X,"-----"
200 IMAGE "CENTRAL CITY SCHOOL DISTRICT"/"DAILY REPORT OF ",25A//
210 IMAGE 20A,20A,"$",DDD.DD,DD.DDD
230 PRINT USING 100
250 INPUT Z
260 READ #Z;A$,N
270 PRINT LIN(6)
500 PRINT USING 200;A$
550 PRINT USING 150
555 PRINT USING 175
557 FOR A1=1 TO N
560 READ #1;B$,C$,A,B
600 PRINT USING 210;B$,C$,A,TAB(50),B
620 NEXT A1
1000 END

```

When executed, the following report is printed:

ENTER SCHOOL NUMBER: ?1

CENTRAL CITY SCHOOL DISTRICT
DAILY REPORT OF B. BAKER HIGH SCHOOL

TEACHER	SUBJECT	SALARY	ATTND.
-----	-----	-----	-----
MISS BROOKS	ENGLISH	\$450.34	12.500
MISS CRABTREE	REM. READING	\$400.00	64.320
MISS GRUNDY	HISTORY	\$350.00	1.001
MRS. HUMPHREY	SPELLING	\$709.00	99.990
COLONEL MUSTARD	CRIMINOLOGY	\$700.00	21.450
MISS PEACH	LIFE PREPARATION	\$232.00	23.235
PROF. PLUM	AGRICULTURE	\$770.90	65.500
MISS H. PRYNNE	SOCIAL STUDIES	\$100.25	
+5.00500E+02			
MISS SCARLETT	P. E.	\$205.10	20.090
MR. WEATHERBY	ECONOMICS	\$767.90	10.040

SYSTEM FACILITIES

SECTION

VII

A set of HP 2000 BASIC statements allows you to control system operation from an executing program:

- The CHAIN statement allows one program to request execution of another. Values can be passed between such programs with the COM statement.
- The IF ERROR statement traps errors that occur in an executing program and returns the error number to the program through the SYS function.
- The SYS function also allows you to programmatically detect whether the BREAK key has been pressed when break is disabled by the BRK function.
- The SYSTEM statement allows a program to execute a set of BASIC commands from the program.

LINKING PROGRAMS

The 2000 system allows you to link programs so that one program can "call" another program for execution. This allows you to segment one large program into several smaller, easier to manage programs.

The CHAIN statement when executed causes the current program to terminate and a referenced program to be brought into your work area and start execution. For example, the following statement, when executed, terminates the program of which it is a part and transfers control to the beginning of another program, PROG2:

```
100 CHAIN "PROG2"
```

Note that the program name is entered as a string literal enclosed in quotes. It could also be a string variable whose value is the program name.

You may precede the program name parameter with a return variable. If for any reason the destination program cannot be executed, the return variable is set to a numeric value indicating the reason for lack of success; if the chain is successful, this variable is zero. The following program illustrates chaining using the return variable:

```
100 CHAIN R,"PROG2"  
110 REM CHAIN FAILED program continues if chain fails  
120 GOTO R OF 130,150,170  
130 PRINT "BAD LINE NUMBER SPECIFIED" R=1  
140 STOP  
150 PRINT "NO ACCESS PERMITTED" R=2  
160 STOP  
170 PRINT "CHAIN NOT PERMITTED" R=3  
180 END
```

When a return variable is not specified, an unsuccessful chain operation terminates program execution.

System Facilities

In the example above, a message is included in case of a bad statement number. This can only occur when the CHAIN statement specifies a particular statement number to which control transfers instead of transferring to the start of the program. Normally, the execution of the destination program starts at the first statement. If you want to start at some other point, you can specify a statement number after the program name. For example:

```
100 CHAIN R,A$,150
```

The program specified in A\$ starts execution at statement 150.

One use of this feature is to allow you to specify a statement number to return to if you want to return to the original program from a chained program. Used in this way, the destination program is similar to a subroutine.

To illustrate, assume a program ALPHA and another program SUBA to which ALPHA transfers. At the end of SUBA, instead of terminating, it returns to ALPHA:

ALPHA

```
10 REM...PROGRAM ALPHA
20 LET X=200
30 LET A=X ** 3
40 PRINT "X CUBED ="A
50 CHAIN R,"SUBA"
60 GOTO 100
70 REM...RETURN HERE FROM SUBA
80 PRINT "CHAIN SUCCESSFUL"
90 STOP
100 PRINT "CHAIN UNSUCCESSFUL FROM ALPHA"
110 END
```

SUBA

```
10 REM...SUBPROGRAM SUBA
20 PRINT "SUBA BEING EXECUTED"
30 CHAIN R,"ALPHA",70
40 PRINT "CHAIN NOT SUCCESSFUL FROM SUBA"
50 END
```

RUN

ALPHA

```
X CUBED = 8.000000E+06
SUBA BEING EXECUTED
CHAIN SUCCESSFUL
```

DONE

This example illustrates how the transfer takes place, but it is clear that it would be more useful if values were passed between programs. This can be done with the COM statement.

PASSING PARAMETERS

The COM statement specifies string or numeric values common to more than one program. This statement must be the first (lowest numbered) statement in any program that either passes or receives values. When COM is used, it can substitute for a DIM statement defining the common values; like DIM, it sets the dimensions of the list of variables. For example:

```
100 COM A$(255),B(10,20),N,A0$(5)
```

The strings A\$ and A0\$ and the array B are dimensioned and defined as common to more than one program; variable N is simply defined as a common variable.

The names of corresponding variables in COM statements need not be the same since equivalence is based on the order in which they appear. You must insure that corresponding variables are of the same type (numeric, string, array) and the same dimensions. To illustrate:

JPROG1

```
10  COM A$(60),B,C,A[2,2]
20  A$="PRINT TWO NUMERIC VALUES FOLLOWED BY A 2 BY 2 ARRAY"
30  B=1.357
40  C=7
50  MAT A=CON
60  CHAIN "JPROG2",100
70  END
```

JPROG2

```
15  COM C$(60),X,Y,Z[2,2]
100 PRINT C$,LIN(1),X,Y,LIN(1)
110 MAT PRINT Z
120 END
```

When JPROG1 is executed, the values assigned in that program are passed to JPROG2 to be printed:

RUN

JPROG1

```
PRINT TWO NUMERIC VALUES FOLLOWED BY A 2 BY 2 ARRAY
1.357          7
1              1
1              1
```

DONE

PROGRAMMATIC ERROR DETECTION

During program execution, error messages normally are sent to the terminal even when they are simply warnings that do not prevent the program from continuing. Sometimes such messages interfere with output from the program. The IF ERROR statement is provided in order to intercept error message display but at the same time keep a record of any execution time errors. This statement may appear anywhere in your program as long as it is executed prior to the statements where you want errors trapped. Once executed, it remains in effect until the execution of a CHAIN, STOP, END, or another IF ERROR statement.

If the error causes only a warning, IF ERROR does not interrupt the sequence of program execution, but if an error is fatal (a programming error, format error, or file error), control branches to a statement specified in IF ERROR.

For example, the following statement could be specified at the beginning of a program:

```
1000 IF ERROR THEN 9000
```

Whenever an error (other than a warning message) occurs subsequent to statement 1000, control branches to statement 9000.

Every error that could occur is assigned an error number (refer to Appendix C). When an error occurs, the error number can be retrieved through the function SYS(0). The line number where the error occurred is returned through function SYS(1). If it was a file error, SYS(2) returns the file number of the last file accessed. If no error has occurred, SYS(0) and SYS(1) return the value zero; if no file has been accessed, SYS(2) returns minus one.

The following example illustrates IF ERROR and the SYS functions.

```
1000 IF ERROR THEN 9000
.
.
.
9000 REM **ERROR TRAPPING ROUTINE**
9005 E=SYS(0)
9010 IF E<100 THEN 9060
9020 IF E<200 THEN 9100
9030 REM
9040 REM ** ASCII FILE ERROR **
9050 PRINT "ASCII FILE ERROR"E"IN LINE"SYS(1)"ON FILE"SYS(2)
9060REM
9060 REM
9070 REM ** PROGRAMMING ERROR **

9055 STOP
9060 REM
9070 REM ** PROGRAMMING ERROR **
9080 PRINT "PROGRAMMING ERROR"E"IN LINE"SYS(1)
9090 STOP
9100 REM
9110 REM ** FORMAT ERROR **
9120 PRINT "FORMAT ERROR"E"IN LINE"SYS(1)
9130 END
```

This example only prints the type of error and stops. Note that the type of error can be determined from the error number. All errors less than 100 are programming errors, errors starting at 100 but less than 200 are format errors. Errors with numbers of 200 and above are either warnings or file errors. Since a warning does not cause transfer to the statement specified in IF ERROR, the example assumes all trapped errors 200 and above are file errors.

PROGRAMMATIC DETECTION OF ABSENCE OF DATA

When using RJE or link terminal devices, the error trapping capability of the HP 2000 system can be used to efficiently control multiple devices.

In the case of link terminal devices, error 300 is returned by the system when an input is requested and no data is present. This feature can be used to write applications collecting data from more than one link terminal. Refer to Appendix G for an example of this application.

In the case of RJE devices, error 300 is returned by the system under two circumstances:

1. when an input is requested and no data is sent
or
2. when output is requested and the IOP is currently unable to accept more data (because the buffer is full).

This feature can be used to write applications that take advantage of the multiple stream capability of the MRJE workstation. Refer to Section 9 for an example of this application.

PROGRAMMATIC DETECTION OF BREAK

Many applications disable the break key with the BRK function in order to prevent a terminal user from terminating an executing program by pressing BREAK. Such applications may want to know if the BREAK key has been pressed. By executing a SYS(3) function, a program can determine if the BREAK key has been physically struck at the terminal when it is disabled. SYS(3) returns a one if BREAK was pressed, a zero if not. It also clears the function so that the next execution of SYS(3) with no intervening breaks will return a zero.

To illustrate:

```

10 A=BRK(0)                                disable break key

                                           statements tested for break

100 IF SYS(3)=0 THEN 140
120 PRINT "BREAK KEY PRESSED, TERMINATE BY OPERATOR"
130 STOP
140

```

At statement 100, a test is made on the preceding statements to determine if break was requested. The program is terminated if so, but continues otherwise. SYS(3) is reset to zero so that subsequent tests for break can be made.

PROGRAMMATIC DETERMINATION OF TERMINAL TYPE

You can programmatically determine the type of the terminal specified with the HELLO command at log on time through the SYS function. SYS(4) returns the terminal type as a digit between 0 and 9. Refer to Table D-1 in Appendix D for a complete list of the terminals associated with each type.

EXECUTING PROGRAM COMMANDS

It is possible to execute some commands from an executing program using the SYSTEM statement. Commands which can be used with the SYSTEM statement are:

GROUP 1

BYE	(logs you off the system)
ECHO	(adjusts data transmission)
MESSAGE	(sends a message to the operator)
FILE	(creates an ASCII file)
PROTECT	(sets a file to the PROTECTED state)
LOCK	(sets a file to the LOCKED state)
PRIVATE	(sets a file to the PRIVATE state)
UNRESTRICT	(sets a file to the UNRESTRICTED state)
MWA	(sets a file to the Multiple Write Access state)
SWA	(sets a file to the Single Write Access state)
PAUSE	(suspends executing program without enabling terminal input)

GROUP 2

TIME	(returns the <i>idcode</i> , <i>port number</i> , amount of current terminal, and total time used and time permitted for your <i>idcode</i>)
CATALOG	(returns a line of entries in your library)
GROUP	(returns a line of accessible entries in your group's library)
LIBRARY	(returns a line of accessible entries in the system library)
LENGTH	(returns the length of the program in your work space, how much space you are using in your library, and the total space permitted in your library)

The commands in group 1 return a numeric value to the SYSTEM statement, whereas commands in group 2 return a string value. SYSTEM requires a return parameter preceding the command it executes. The command itself is specified as a string constant or variable. Note that the PAUSE command is unique in that it does nothing unless executed through the SYSTEM statement.

When the SYSTEM statement is used with a command in the first group, it uses a command string and a numeric return variable as parameters. The command string is the command to be executed. This may be a string variable containing the command or the command itself. The form of the command is exactly the same as it would be if you had entered it from your terminal. The return variable is used to indicate whether or not the command executed properly. The return variable is set to "0" for successful execution or to "1" if the command could not be executed.

To illustrate, send a message to the system console from an executing program:

```

5  DIM A$(40)
10 A$="MESSAGE-HELLO OUT THERE"
20 SYSTEM R1,A$
30 IF R1=0 THEN 50
40 PRINT "MESSAGE WAS NOT SENT"
50 END

```

A message is sent to the user terminal only if the command is unsuccessful.

An ASCII file can be created only by the FILE command. To create an ASCII file from a program, the SYSTEM statement must be used:

```

10 SYSTEM R,"FILE-LPRTR,LP0"
20 IF R=0 THEN 40
30 PRINT "FILE LPRTR NOT CREATED"
40 STOP
100 END

```

One use of the PAUSE command is to pause and try again. Suppose a device is busy, a transfer to the following subroutine will pause for 60 seconds and then try the device again:

```

100 GOSUB 250
250 REM...PAUSE ROUTINE
255 SYSTEM R3,"PAUSE-60"
260 IF R3=0 THEN 270
265 PRINT "PAUSE NOT EXECUTED"
270 RETURN

```

When the SYSTEM statement is used with a command in the second group it uses a return or destination string instead of a numeric return variable. Each of the commands in the second group would normally produce one or more lines of output at your terminal. When used in the SYSTEM statement only the first line of output (less any heading) is returned in the destination string.

For example, to retrieve as a return string, the string returned by the LEN command:

```

10 DIM A$(72)
20 SYSTEM A$,"LEN"
30 PRINT A$
40 END
RUN

```

```

00019 WORDS = 01 RECORDS. 01551 RECORDS USED OF 65000 PERMITTED.

```

```

DONE

```

System Facilities

When more than one line of output is required you can obtain successive lines of output by modifying the command. In the following example, the LIBRARY command is used to obtain a list of system programs and files. Each time a line of output is received, the starting name used in the command is modified to obtain the next sequential entry.

```
10 DIM A$(10),B$(72)
20 A$="LIB-A"
30 SYSTEM B$,A$
40 PRINT B$
50 A$(5,10)=B$(49,54)           assign last library entry on line
60 FOR I=10 TO 5 STEP -1       to LIB command
70 IF A$(I,I)="9" THEN 110
80 IF A$(I,I) <> "Z" THEN 130   increment last entry to get next line
90 NEXT I
100 GOTO 30
110 A$(I,I)="0"
120 GOTO 30
130 A$(I,I)=CHR$(NUM(A$(I,I))+1)
140 GOTO 30
150 END
RUN
```

ACF2	U	2	ACF3	U	2	ACF4	U	2
ACF5	U	6	ACF6	U	4	ACKERM	FP	4
AEVS	U	4	ANAL1	U	5	ANAL2	U	4
STOP								

The same type of code could be used with either the CATALOG or GROUP commands. To start at a particular item, line 20 could be modified and another line added as follows:

```
20 A$="LIB-"
25 INPUT A$(5,10)
```

This allows you to enter the characters at which you want the list to start. Rather than printing the entire list or terminating it with the BREAK key, you could put a terminating entry in the program. For example

```
30 IF A$(5,10) ="TEST09" THEN 150
35 SYSTEM B$,A$
```

These lines of code will cause the list to terminate when a value greater than or equal to "TEST09" is reached.

SECURITY AND THE LIBRARY HIERARCHY

SECTION

VIII

USER IDCODE ORGANIZATION

The system utilizes identification codes (idcodes) to identify every user. A unique password may be used along with each idcode to insure account security; but the system does not actually require its use. The idcodes are organized by the system for accounting purposes, controlling accessing capabilities, data security and account privacy. An idcode is made up of a single alphabetic character followed by three decimal digits. The range is from A000 to Z999 (inclusive); making a total of 26,000 distinct idcodes.

The system groups idcodes, 100 to each group. For example, A000 through A099 constitute the first group, A100 through A199 constitute the second group and so on through the last possible group, Z900 through Z999. There is a total of 260 groups.



User organization (by idcode) is illustrated in figure 8-1. The three levels represent the three types of users as follows:

1. The very first idcode within the system, A000, is assigned to the user who will serve as the system master.
2. The first idcode within each group (for instance A100, D300, etc.) is assigned to the user who will serve as group master.
3. All other idcodes are available for assignment to private users.

Since A000 is both the first idcode in the system and the first idcode of the first group, the system master serves as group master for the first group. (Duties of system and group masters are discussed later.)

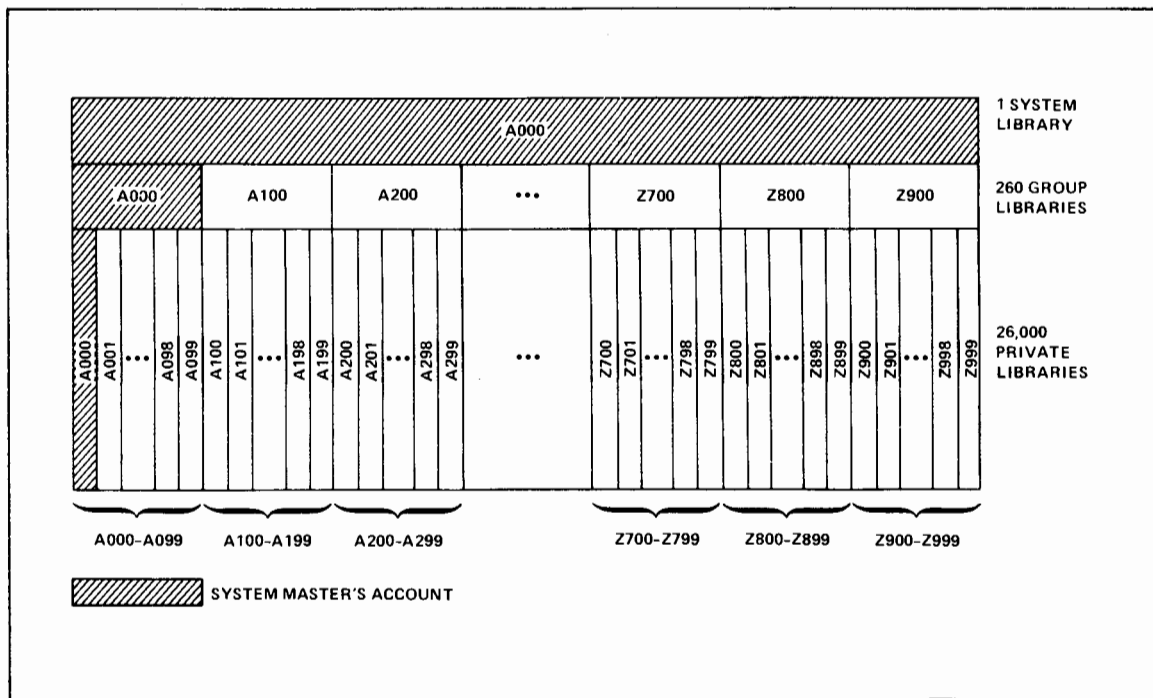


Figure 8-1. Idcode/Group Account Structure

When an idcode is created, an upper limit is set to the amount of disc space permitted to the idcode for storing programs and files. As the user allocates disc space by storing programs and creating files, he creates a "library" for his idcode.

Just as there are three types of users, so are there three types of libraries associated with those users.

1. The library assigned to the system master is referred to as the system library.
2. A library assigned to a group master is called a group library.
3. The library of every other user is known as a private library.

In figure 8-1, the lower layer of idcodes represents private libraries, the middle layer of idcodes represents the group libraries, and the upper layer represents the system library. (Note that the system library is also the group library for the first group and the private library for account A000, the system master's account). Each user can access elements in his own private library and, to the limit of any restrictions, programs and files in his group library and in the system library. A user cannot access other libraries unless these libraries have been given Program/File Access (PFA) capability. (Refer to Account Accessing below.)

PRIVATE LIBRARY — PRIVATE USER

A private library is created and maintained by each idcode. This library is completely controlled by the user assigned that idcode. The user can enter, modify, restrict access to, and delete programs and files within his private library. When the account has Program/File Access (PFA) capability, its library can (under certain circumstances) be accessed and altered by the group master or by other users. A list of programs and files in a user's library is printed by executing the CATALOG command.

GROUP LIBRARY — GROUP MASTER

A group library is the library of the group master. The group master may remove any accessing restrictions on individual programs and files in the library so they are accessible to members of the group. If the idcode has Program/File Access (PFA), all or part of the library can be made accessible to all other system users. The group master is responsible for creating, maintaining, deleting, and controlling access to the programs and files within this library. Programs residing in the group master's library are permitted to create, give Multiple Write Access (MWA) status to, read, write, and purge locked BASIC Formatted files in the accounts of group members with PFA, but only if the group master has been given the File Create/Purge (FCP) capability by the system operator. A list of the programs and files in the group library accessible to group members is printed by the GROUP command.

SYSTEM LIBRARY — SYSTEM MASTER

The system library is the library of the system master. It is a common library normally available to all users of the system. The system master may remove accessing restrictions on individual programs and files within this library thereby making them available to users other than the system master. The system library programs and files available for general use may be listed with the LIBRARY command.

Basic functions of the system master are similar to those of a group master, but more extensive. The system master can enter, modify, restrict access to, and delete programs and files in the system library. The system master is also the group master for accounts A001 through A099 and, as such, can be given File Create/Purge capability over their libraries.

The system master is responsible for creating and maintaining the optional HELLO program within the system library. This BASIC language program executes whenever you log on. Typically, the HELLO program provides such information as:

- System identification
- Port Number
- Date and time of log on
- System hibernate and sleep schedules
- News for the day
- News about new applications programs

It is possible to use the HELLO program to create a dedicated environment. If an account is to execute only one particular application program, and not be permitted any other system activity, the HELLO program can channel the user to the application program immediately upon log on. This might be accomplished in the following manner:

- a. Establish a file that contains (for each account on the system) information about planned account usage. This file should reside in A000 and be locked; hence only available to locked system library programs.
- b. Write the HELLO program so that it determines the user's log on idcode, scans the file previously established, and takes the user to the appropriate program. Or the HELLO program could CHAIN to some other program that performs this task.
- c. The application program should disable the BREAK key at the user's terminal and execute a BYE command whenever it relinquishes control. Note that the HELLO program always runs with the BREAK key initially disabled but it can re-enable the BREAK key.

ACCOUNT ACCESSING CAPABILITIES

Thus far we have discussed the accessing capabilities the system automatically grants each idcode. However, there are other types of accessing capability not associated with the idcode/group structure. Three special accessing capabilities can be granted when the system manager assigns or changes an idcode. Two of these (PFA and MWA) may be granted to any idcode; the third (FCP) is meaningful only if granted to a group master's idcode. Recall that the system master is also a group master. These capabilities are summarized in Table 8-1.

Table 8-1. Account Accessing Capabilities

LIBRARY	SELECTED CAPABILITY	DEFAULT CAPABILITY	DESCRIPTION
User's Library	PFA	NOPFA	Program/File Access — allows a user to make selected programs and files in his library accessible to all other users, thereby forming a pseudo system library. This allows a user to access libraries other than his own, his group library, and the system library.
	MWA	SWA	Multiple Write Access — allows a user to give MWA status to selected files in his library so that several users can write to these files simultaneously. The default, Single Write Access, allows only one user at a time to write to the file.
Group Master's Library	FCP	NOFCP	File Create/Purge — provides a group master with special accessing capabilities within the libraries of his own group members. A locked or private program in the group master's library may: create, assign MWA status to, read, write, or purge locked BASIC formatted files, or it may chain to a line number in a locked program. The library containing these files or programs must be a member of the group and must have PFA status. The program in the group master's library that performs these functions must be executed with the EXECUTE (not the RUN) command or be chained to by a CHAIN statement.

The PFA, MWA, and FCP capabilities may be assigned by the system operator commands NEWID and CHANGEID; once assigned, the default capabilities can be selected with CHANGEID. (Refer to the HP 2000 System Operator's Manual for a description of these two commands.)

Considering idcode/group accessing capabilities and the PFA, MWA, and FCP capabilities together with the protected, locked, unrestricted, and private states of programs and files (described below under Program and File Restrictions), it is possible to plan elaborate data

security and access schemes for your applications. Remember that the ninety-nine accounts A001 through A099 are unique since they have the system library as their group library. This feature might be utilized in a number of ways. For example, an applications program can be a locked program in the system library which in turn has the FCP capability. As such, it can programmatically create, manipulate, and purge files in the A0xx accounts, even when being run from some other account on the system.

PROGRAM AND FILE RESTRICTIONS

When Program/File Access (PFA) has been granted to an account, each program and file saved in that library may be accessed if the status of the program or file allows. There are four states in which a program or file may be saved that determine their access status: private, locked, protected, and unrestricted.

PROGRAM STATES. Figure 8-2 illustrates the allowable access to a program in your account by users logged on in accounts other than yours.

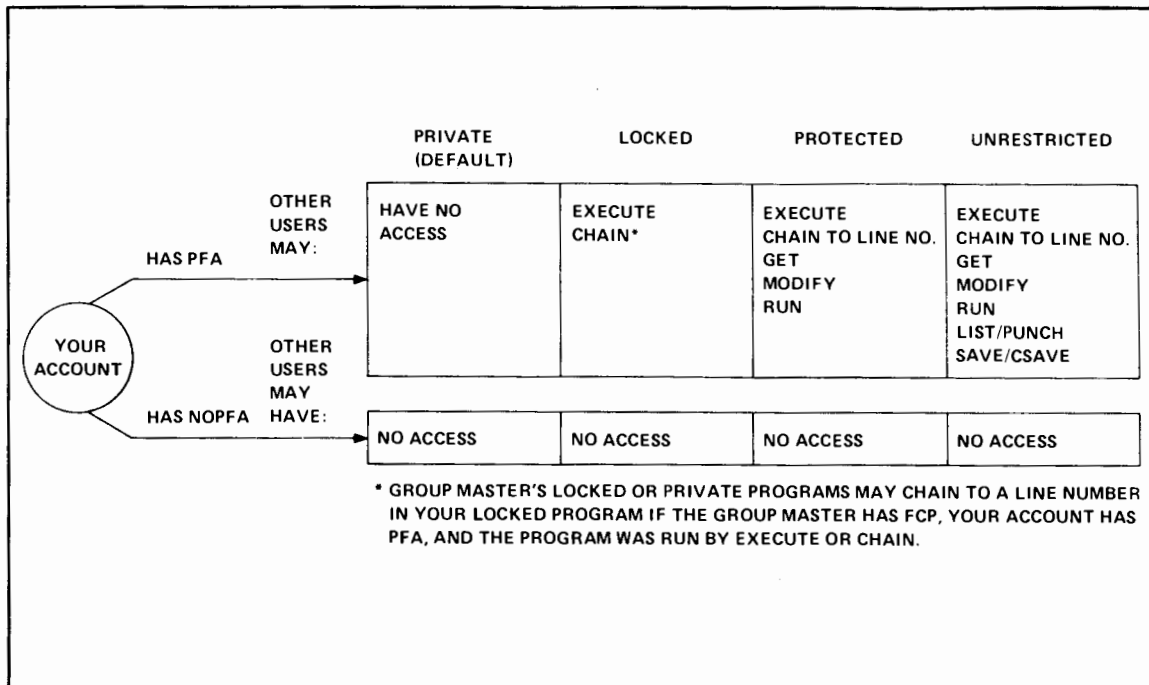


Figure 8-2. Program Access States

When a program is saved, it is automatically placed in private status. To assign it another status, you use one of the commands LOCK, PROTECT, or UNRESTRICT. To return a program to private status, you use the PRIVATE command.

When a library is listed with one of the commands CATALOG, GROUP or LIBRARY, any csaved programs are followed by the program descriptor C; saved programs have a blank program descriptor. If an access restriction has been placed on the program, it is indicated by a U for unrestricted, P for protected, or L for locked. Private programs are followed by a blank and are not listed by the GROUP and LIBRARY commands.

If you want to guarantee a program absolute privacy from other users, you should leave it in the private state in which it was saved. A private program can be referenced only by you or a user with your idcode.

Locked programs are useful when you want to allow other users to run your program in its entirety but not be able to list or modify it. Locked programs are also useful for accessing locked files. Your locked program has read and write access to your locked files even when your program is executed by another user. Your group master's locked or private programs can chain to a line number within your locked program, but only if:

- 1) your group master's account has the FCP capability.
- 2) your account has the PFA capability.
- 3) the group master's program was executed by an EXECUTE or CHAIN statement.

A protected program is useful in applications where one user supplies the program and other users supply their own data in the form of DATA statements appended to their copies. Another application is to protect a program used as a subroutine by another user's main program, thereby allowing users to append the subroutine to the main program but not list it.

When a program is unrestricted, any other user may get, list, save, or modify his copy of the program.

FILE STATES. Figure 8-3 illustrates the allowable access to files in your account by users logged in to accounts other than yours.

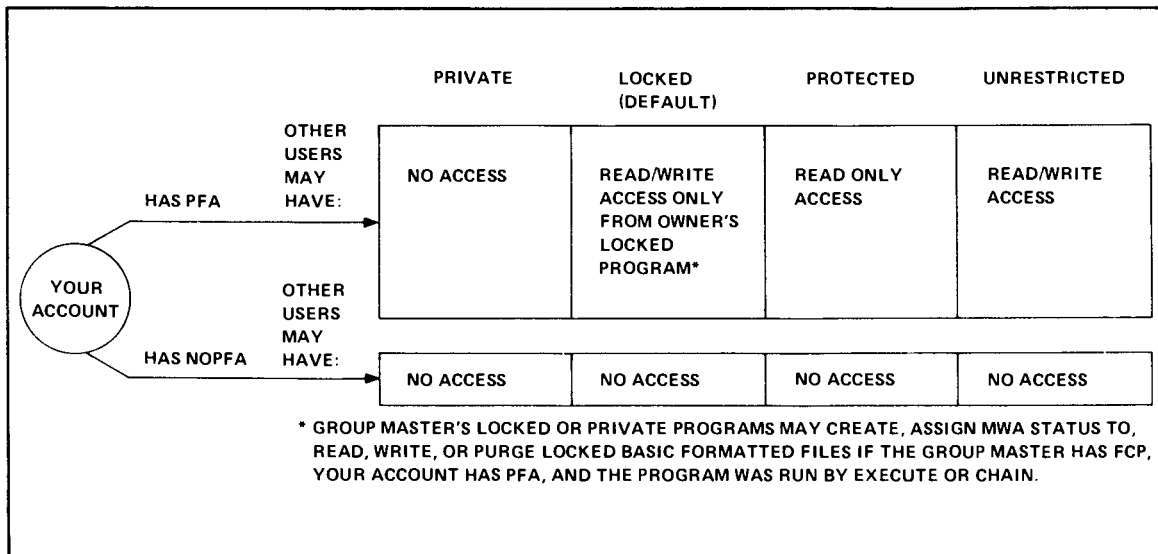


Figure 8-3. File Access States

When a file is created, it is automatically placed in locked status. To assign it another status, you may use one of the commands PRIVATE, PROTECT, or UNRESTRICT. To return it to locked status, use the LOCK command.

When a library is listed by a CATALOG, GROUP, or LIBRARY command, each file name is followed by either an A for an ASCII file, an F for a BASIC Formatted file, or an M for a BASIC Formatted file with Multiple Write Access. Locked, protected, and unrestricted files are indicated by an L, P, or U respectively. Private files have a blank access restriction code and are not listed in the group or system library listings.

A private file may be accessed only by you or a program executed through your idcode.

Your locked files can be read or written to by a locked program in your library regardless of who is executing that program, but your locked files are otherwise unavailable to users not logged in to your account. There is one exception: Your group master's locked or private programs may create, read, write, purge, or assign MWA status to your locked BASIC formatted files, but only if:

- 1) your group master's account has the FCP capability
- 2) your account has the PFA capability
- 3) the group master's program was executed by an EXECUTE or CHAIN statement.

ASCII files equated to non-sharable devices that belong to one user's account cannot be accessed by another account unless the file is locked, resides in the system library, and is accessed by a locked program also resident in the system library. For example:

HEL-A000,AA	
FIL-READR,CR,40	READR is locked ASCII file belonging to A000
HEL-X123,BB	
EXE-\$EDITR	EDITR, a locked program in A000, can access READR

It is only meaningful to give protect status to BASIC formatted files since ASCII files are by definition non-sharable devices. Files that are protected may be read, but not written to or updated. Protecting a file allows you to set up a data base that can be read by other users but only updated by you.

As with protected files, the unrestricted state is only meaningful for BASIC formatted files since ASCII files are non-sharable devices.

CONTROLLING SIMULTANEOUS WRITING ON FILES

A user logged on under an idcode that has been assigned MWA capability can give multiple-write-access to any files in his account through the MWA command or remove this capability with the SWA command. Single-write-access (SWA) is the default for all files.

Multiple-write-access means that a file can be written to by more than one user concurrently. In order to control such access, two statements are provided: LOCK and UNLOCK

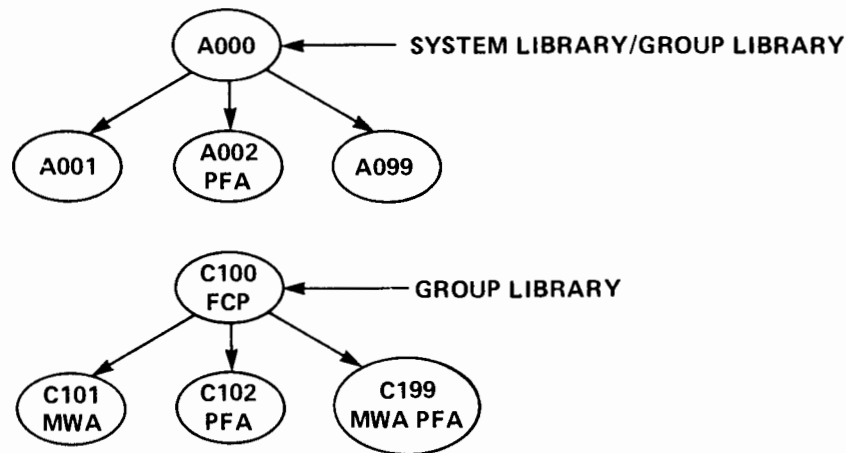
Cooperating users may use the LOCK and UNLOCK statements to prevent writing on the same record at the same time, which might otherwise invalidate the write operation. The LOCK and UNLOCK statements do not actually "lock" or "unlock" a file. They set and clear

“busy flags” associated with the named file. Execution of the LOCK and UNLOCK statements return the current status of this flag.

When you want to write on a MWA file which may already be in use, you should first lock the file. If another user already has locked the file, your lock will not take effect until that user unlocks the file. Cooperating users can then insure that they are not all trying to write on the file at the same time. (Do not confuse the statement LOCK with the locked state. The former is a facility available from a running program to prevent simultaneous writes; the latter is a state the file may be in which restricts some kinds of access to the file.)

Examples

To illustrate some of the concepts discussed earlier, assume the following accounts:



PFA has been assigned to account idcodes A002, C102, and C199.

MWA has been assigned to C101 and C199.

FCP has been assigned to group library idcode C100.

- As a result of giving PFA to accounts A002 and C102, any non-private file or program in one of these accounts can be accessed from any library in the system by appending the idcode to the program or file name. For example, assume PROGA is in A002, it can be executed from C199:

```
HEL-C199
EXE-PROGA.A002      A002 has PFA; PROGA may not be private.
```

- As a result of giving MWA status to C101, a program run from idcode C101 can assign MWA status to any file in that library. Assume file FILE1 is in C101 library, to assign MWA status to that library:

```
HEL-C101
MWA-FILE1           C101 has MWA
```

Note that the command MWA can be used only in accounts that have been given MWA status.

3. A locked or private program executed from group account C100 which has been assigned FCP may perform special functions on files or programs in accounts C102 and C199 which have PFA. Note that the idcodes with PFA must be within the group library; C100 can perform no special functions in account A002.

To illustrate, create file CTEST, assign it MWA status and then chain to the locked program PROGC at line 500; PROGC must be in a Cinn library with PFA (in this example, idcode C102).s, must be in a C1nn library with PFA (in this example, idcode C102).

```
HEL-C100          C100 has FCP
EXE-PROG2        EXE (not RUN) must be used to run a program using FCP
PROG2

10 CREATE R,"CTEST.C199",30      create CTEST in account C199
20 SYSTEM R,"MWA-CTEST.C199"    give CTEST MWA status
30 IF R=0 THEN 50
40 PRINT "CTEST NOT ASSIGNED MWA" error message
45 STOP
50 CHAIN X,"PROGC.C102",500
60 IF X=0 THEN 80
70 PRINT "CHAIN TO PROGC NOT SUCCESSFUL"
80 END
```

DATA COMMUNICATIONS

SECTION

IX

This section presents an overview of the data communications facility of your HP 2000 system as provided through:

The Remote Job Entry (RJE) facility which transmits jobs between an HP 2000 system and a remote host system. The HP 2000 emulates one of the following:

- Multileaving RJE Workstation (MRJE W/S)
- 2770/2780/3780 terminal
- User 200 terminal

2000 to 2000 Communication which transmits data between one HP 2000 system and another.

2780 to 2780 Communication which transmits data between an HP 2000 system emulating a 2770, 2780 or 3780 terminal, and one of the following:

- 2770, 2780 or 3780 terminal (with the same configurations as the Emulator)
- another HP 2000 system running the 2770/2780/3780 Emulator
- an HP 3000 system running the 2780/3780 Emulator
- any other computer system emulating a 2770, 2780 or 3780 terminal (with the same configurations as the Emulator)

WHAT IS REMOTE JOB ENTRY?

Remote Job Entry is a system facility that allows you to submit a job from the 2000 system for processing on another system known as the host. When sending jobs to a host system, the 2000 system acts as a remote terminal or input device for the other system; when receiving output from jobs, the 2000 system acts as a terminal or output device for that system.

Programs contained in jobs may be written in any language (FORTRAN, COBOL, RPG, BASIC, etc.) available on the host system. Programs may be assembled, compiled, or executed on the host system. The programs can retrieve data files from the host system and send them back to the 2000 system. The results of processing can be returned to the local 2000 system for further processing, storage, or output, or processing results may be routed to peripheral devices associated with the host system.

The RJE facility transfers data over the public telephone network or private leased lines at rates up to 9600 bits per second (approximately 960 characters per second). A functional diagram of the system elements used by the RJE facility is given in figure 9-1.

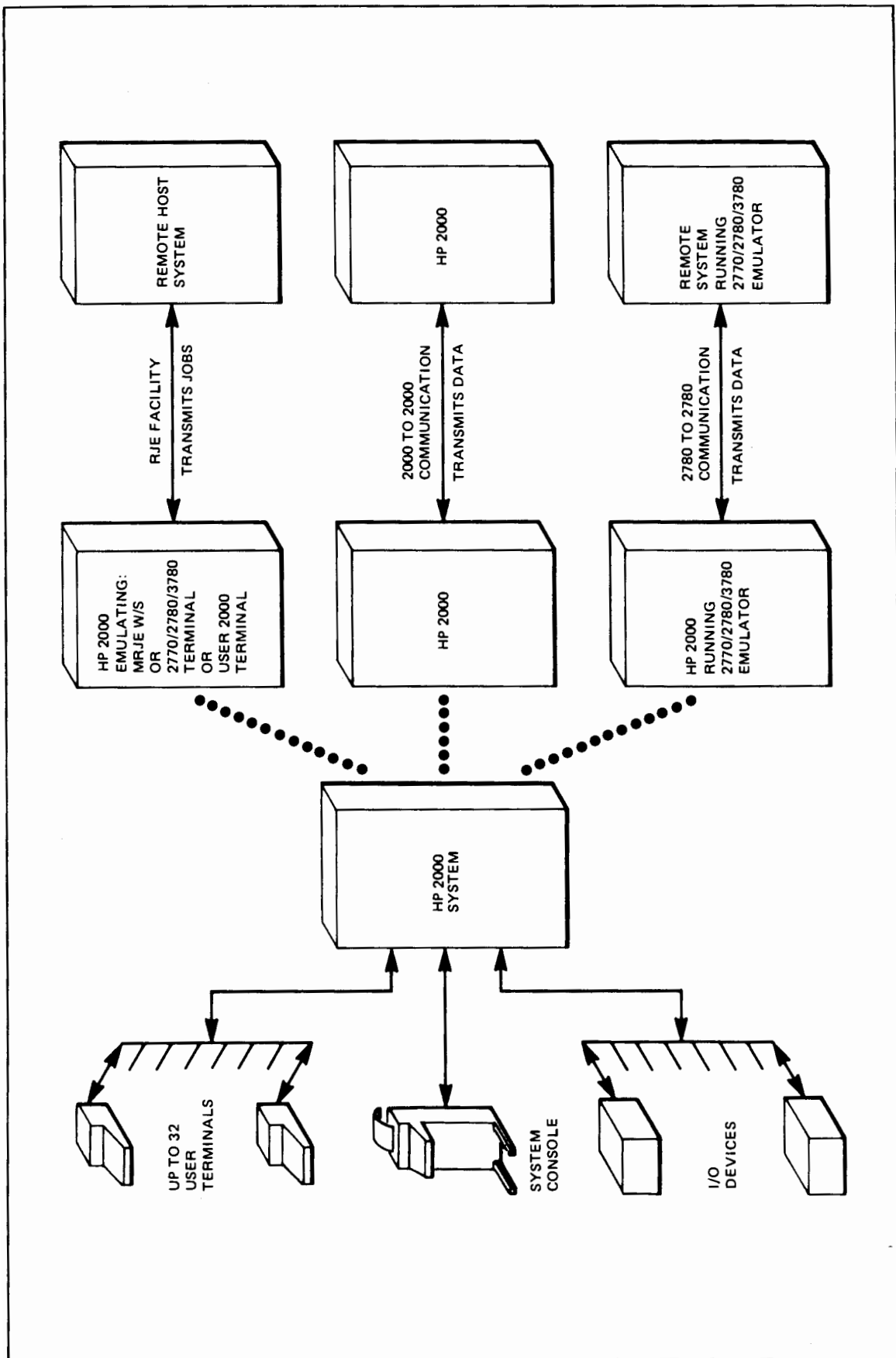


Figure 9-1. Elements of an HP 2000 Data Communications Facility

WHAT HOST SYSTEMS CAN YOU COMMUNICATE WITH?

Table 9-1 lists the host computer operating systems that can be accessed by the RJE facility.

Table 9-1. Host Operating Systems Compatible with RJE Facility

RJE Emulator	Host Systems
Multileaving RJE Workstation	IBM 360/370 OS/MFT/HASP OS/MVT/HASP OS/MVT/ASP OS/VS1/JES/RES OS/VS2/JES2 OS/VS2/JES3
2770/2780/3780 Terminal	Any host system that supports a 2770, 2780 or 3780 terminal, as long as the host system is compatible with the exact terminal configurations emulated. (Refer to Table 9-9.)
User 200 Terminal	CDC 3300/6400/6600/Cyber 70 KRONOS/EXPORT/IMPORT SCOPE/INTERCOM

Tables 9-4 and 9-5 summarize IBM HASP and ASP commands; Table 9-6 summarizes the CDC EXPORT/IMPORT commands; and Table 9-7 lists useful IBM and CDC manuals. Refer to the appropriate host system reference manual for a list of remote commands if you are using the 2770/2780/3780 Emulator.

MULTILEAVING RJE WORKSTATION (MRJE/WS)

When the RJE facility is used with an IBM host system, listed in Table 9-1, it emulates an MRJE workstation. An MRJE workstation is a remote batch input/output port operating under the multileaving protocol. The remote port uses several data streams or functional lines for simultaneous input, output, and control communication. Table 9-2 contains a list of these functions. There are seven host reader functions for accepting input; seven host list functions for producing output; seven host punch functions for producing punched output; a host inquiry function for entering system requests or commands, and a host message function for receiving responses to system commands and system messages.

Table 9-2. IBM MRJE Workstation Host Functions

HI1	Host Inquiry, used to send commands to the host system
HM1	Host Message, used to receive messages from the host system
HR1-7	Host Reader, used to enter jobs, normally a card reader
HL1-7	Host Lister, used to output jobs, normally a line printer
HP1-7	Host Punch, used to output jobs, normally a punch

2770/2780/3780 TERMINAL

When the RJE facility is used with a host system that supports IBM 2770, 2780 or 3780 terminals, it emulates one of these terminals. The 2770/2780/3780 emulator is similar to the MRJE workstation described previously, but uses only one host reader and one host lister. There are no host inquiry functions, host message functions, or host punch functions.

If the host system is an IBM operating system, and a choice exists between emulating an MRJE workstation and a 2770, 2780 or 3780 terminal, the MRJE workstation provides a more efficient choice.

USER 200 TERMINAL

When the RJE facility is used with one of the CDC host systems listed in Table 9-1, it emulates a User 200 Terminal. The User 200 Terminal is similar to the MRJE workstation described previously but uses only one host reader, one host lister, one host inquiry, and one host message function (HR1, HL1, HI1, HM1). CDC does not provide a host punch function.

HOW DOES RJE WORK?

When you use the RJE facility, portions of the 2000 system act like a remote port to the host system. The various host functions are assigned to various real or virtual devices. The RJE facility looks like a collection of input and output devices to the host system. A typical configuration for an IBM host system used with an MRJE workstation emulator is shown in figure 9-2. Note that some of the host functions have been assigned to virtual devices or job function designators.

The exact number of host functions assigned by IBM host systems can be varied. However, only one host function of each type is used with CDC systems (including the host inquiry and host message functions) while only one host reader and one host lister are used with the 2770/2780/3780 Emulator. The host inquiry and host message functions are not used with the 2770/2780/3780 Emulator. A table of job function designators is given in Table 9-3. The job functions correspond to the host function types.

Table 9-3. Job Function Designators

JOB FUNCTION DESIGNATOR	DESCRIPTION
JIO	Job Inquiry Designator, allows you to send commands and messages to the Host Inquiry function in an executing program.
JMO	Job Message Designator, allows you to read messages from the Host Message function in an executing program.
JT0-6	Job Transmitter Designator, allows you to send jobs to a Host Reader function in an executing program.
JL0-6	Job Lister Designator, allows you to read the output of a job from a Host Lister function in an executing program.
JPO-6	Job Punch Designator, allows you to read job output from a Host Punch function in an executing program.

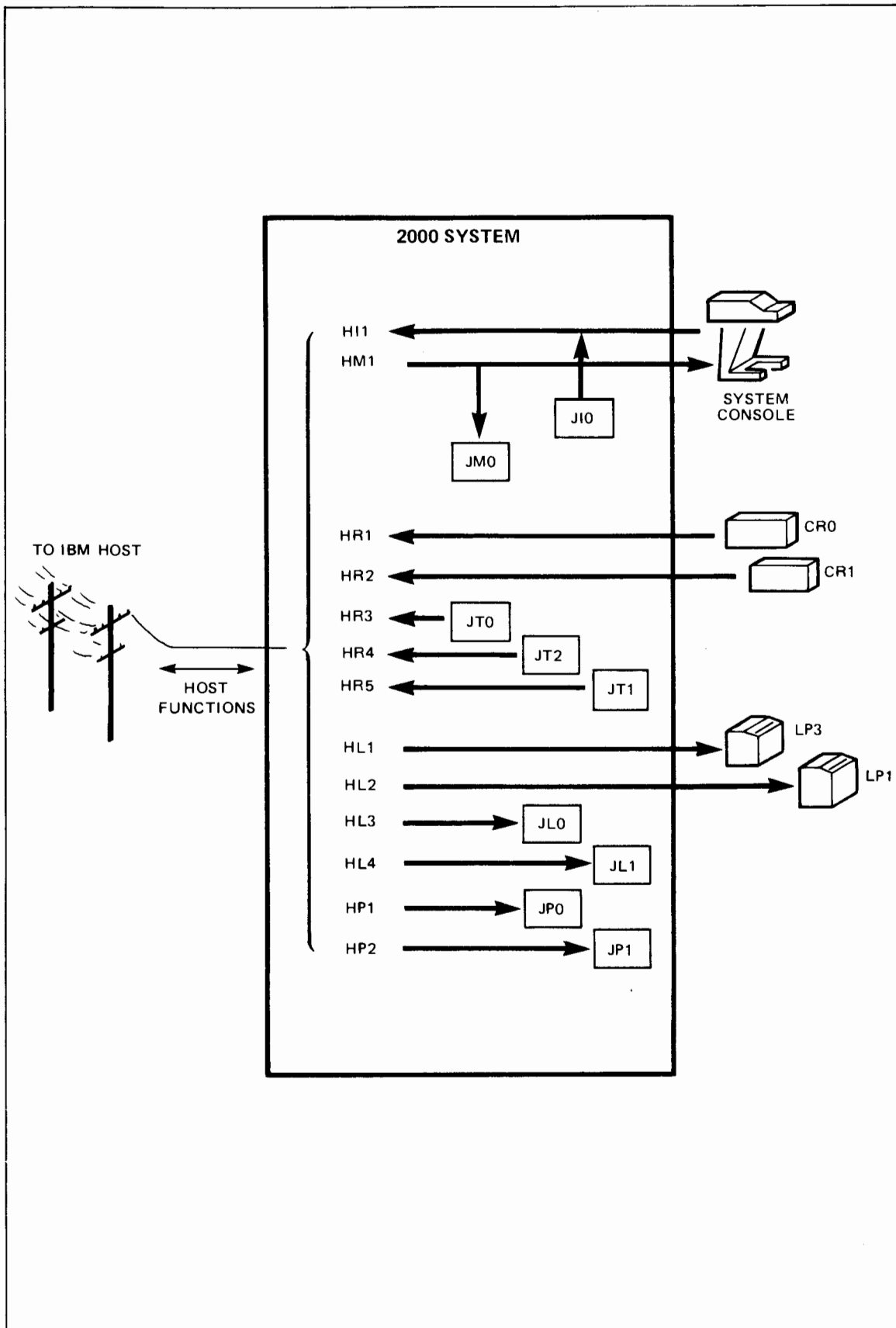


Figure 9-2. Typical RJE Configuration For An IBM Host System Using The MRJE Workstation Emulator.

These job function designators look like real devices to the host system but in fact pass input and output data to programs executing on the 2000 System. This feature allows Remote Job Entry to be accomplished easily by a 2000 system user from his terminal.

When the 2000 system is configured, each of the host functions (HR, HL, HP) allocated to the remote port must be assigned to either a real device (card reader, line printer, paper tape punch) or to a job function designator (JT, JL, JP). If the host inquiry and host message functions are used, they are always assigned to the 2000 system console. It is optionally possible to assign the host inquiry function (HI) to a job inquiry function (JI) and the host message function (HM) to a job message function (JM). Other than the host inquiry and message functions, the assignments may be changed at any time by the system operator.

HOW DO YOU USE REMOTE JOB ENTRY?

When you are ready to use the RJE facility, the system operator typically dials up the host system connecting the RJE modem. The operator then enters a sign on command at the system console followed by any special configuration commands that may be required for a given job. Once this has been done the RJE facility is ready to accept input from card readers or from BASIC programs via the job transmitter function. Any output ready to be returned from the host system can be sent to printers, punches, or to BASIC programs through the job list or punch functions.

Once the communications link has been established and the operator has signed on to the host system, the system console can be used to enter commands to be executed by the host system. These commands can be used to obtain job status, modify the configuration of the remote port, change job priority, or change the routing of job output. The commands available vary with different host systems.

When using the IBM MRJE workstation emulator or CDC User 200 Terminal emulator, you can also send host commands from an executing program by printing on the job inquiry file; and receive any host system responses by reading the job message file.

SENDING JOBS THROUGH THE CARD READER

An easy method of sending a job to the host system is to send a card deck through a card reader. The card reader must be connected to a host function HR (Host Reader). In this case all that is required is to load the card deck in the card reader and start the reader. When the RJE facility is enabled each reader assigned to RJE is automatically made ready. Your job is automatically sent to the host system for processing. Figure 9-3 shows a typical RJE card deck for a FORTRAN compile and execute on an IBM system. To indicate the end of the job to the 2000 system, the last card in the deck to be read must be punched with : : in the first two characters.

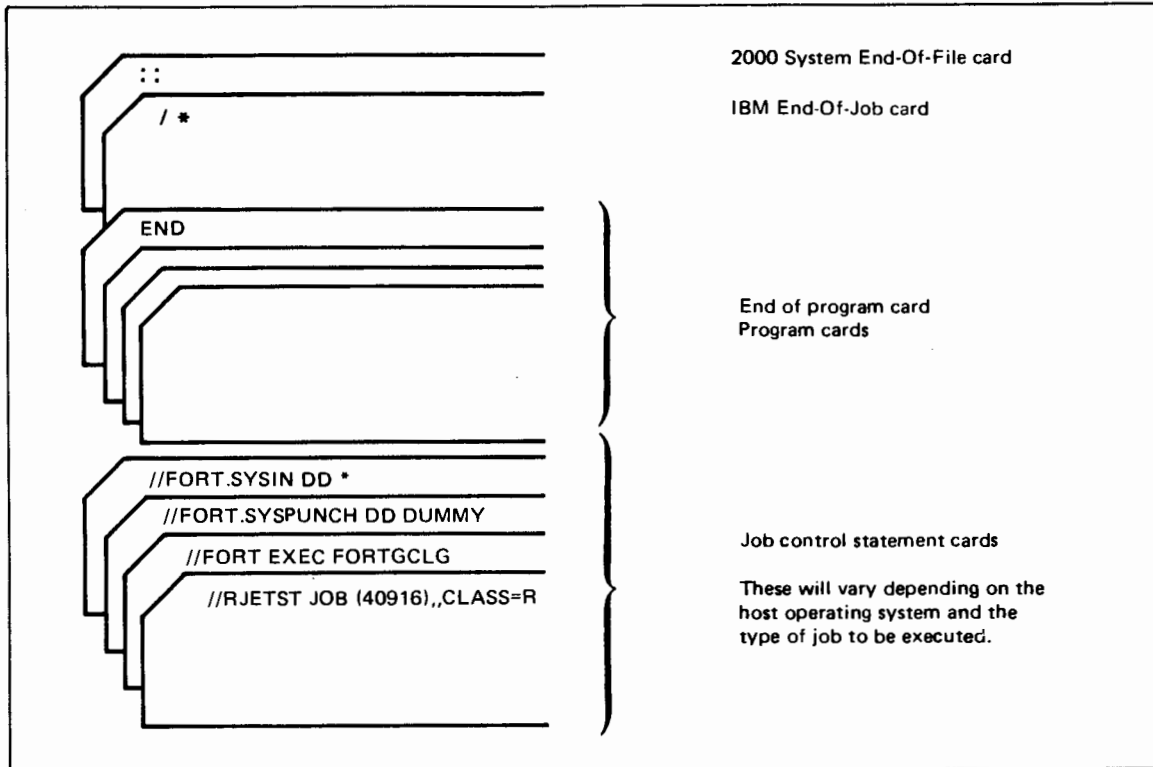


Figure 9-3. Example of an RJE Job Deck

RETRIEVING OUTPUT ON THE LINE PRINTER

When the host system finishes processing a job, the job's output is directed back to the 2000 system on a host function HL (Host Lister) line. If a line printer is connected to the host function HL, the job output is directed to the line printer. When using the CDC User 200 Terminal Emulator and the 2770/2780/3780 Emulator there is no problem determining to which host lister function the output will return because these systems can have only one host lister function. IBM systems working with an MRJE workstation emulator normally return job list output to the first available host list function and job punch output to the first available host punch function.

If you are using the IBM MRJE workstation emulator and want to ensure that output is directed to a specific host function, you must use forms control commands. The IBM host systems allow you to designate specific host functions for certain output formats. This is done using the 2000 system operator's RJE commands to send a forms command through the host inquiry function. The use of forms control is discussed later in this section under the heading "HOW TO GET YOUR OUTPUT." If you are not concerned about the specific line printer or punch to which job output is sent, then you need not be concerned with forms control.

SENDING JOBS AND RETRIEVING OUTPUT THROUGH A PROGRAM

Host reader, list, and punch functions (HR, HL, HP) need not be connected to actual devices. Instead, they may be connected to job function designators. In turn, a job function designator may be equated to an ASCII file name. A BASIC program manipulates these job functions just as if they were non-sharable devices. You may print to a job transmitter (JT), or read from a job lister (JL) or job punch (JP).

Sending a job from your terminal involves two steps. First you must determine what host functions are connected to which job function designators. Your system operator must provide this information as it differs from system to system. To make the following discussion easier, we will assume that you have access to all five types of job function designators and that they are connected as follows:

J10 →HI1	Send messages and commands to host
JM0 ←HM1	Receive messages from host
JT0 →HR1	Send jobs to host reader one
JT1 →HR2	Send jobs to host reader two
JL0 ←HL1	Receive printed output from host lister one
JL1 ←HL2	Receive printed output from host lister two
JP0 ←HP1	Receive punched output from host punch one

After determining what host functions are connected to which job function designators, you are ready to transmit jobs from your terminal. As with any other non-sharable device you must first equate an ASCII file name to a job function designator.

You may find out what job function designators you are allowed to use through the DEVICE command. If you attempt to run a program which accesses a job function designator for which no RJE connection has been established, you will receive the message JT0-DEVICE NOT READY (substitute for JT0 the job function designator used). You may then abort your program with the BREAK key or ask the operator to establish RJE communications. Once established, your program continues automatically unless you abort it.

The job function designators are assigned to ASCII file names with the FILE command. The maximum record size for any device is printed by the DEVICE command. The maximum size for JT devices depends on your system configuration, but can be no more than 66 words (132 characters). If more than 40 words (80 characters) are transmitted to an IBM or CDC host, the excess characters are stripped. For efficiency, when transmitting to IBM or CDC, a file name should be equated to JT with 40 word records. For example, the following command equates file SEND to a JT device using a maximum of 40 words (80 characters):

```
FIL-SEND,JT,40
```

To simulate a CDC 7/8/9 (end of record) card, you can print a line with the numeric equivalent of an ASCII record separator (RS) character. Either of the following statements is equivalent to a 7/8/9 card:

```
100 PRINT #1;'30
100 PRINT #1;CHR$(30) } sends 7/8/9 equivalent to file number 1
```

Similarly, the CDC 6/7/8/9 (end of file) card can be simulated by the numeric equivalent of an ASCII file separator (FS) character. For example:

```
110 PRINT #1;'28
110 PRINT #1;CHR$(28) } sends 6/7/8/9 equivalent to file number 1
```

You can now write a program to send a job. All that is necessary is to retrieve the lines that compose the job (from your terminal, data statements, other files, etc.) and to print them on a file equated to the job transmitter device.

Note that within the BASIC program executed in the local 2000 system, you always write to the JT (job transmitter) devices and read from the JL (job lister) and JP (job punch) devices. Refer to Figure 9-2 for clarification of the function of job function designators.

Example:

The following program sends the job in the DATA statements to an IBM host system.

```
FILE-SEND,JT,40
LIST
```

10	FILES SEND	Opens job transmitter file
20	DIM A\$(80)	
30	IF TYP(0) <> 3 THEN 50	If no more DATA statements,
40	STOP	Stop
50	READ A\$	Read a line of the job
60	PRINT #1;A\$	Send it to the host
70	GOTO 30	Loop

Actual Job to be sent

```
1000 DATA "//SENDJOB JOB (1234567,10,0),'JEC'"
1010 DATA "// EXEC PGM=IEBGENER"
1020 DATA "//SYSIN DD DUMMY"
1030 DATA "//SYSPRINT DD SYSOUT=A"
1040 DATA "//SYSUT1 DD DSN=D0911274.ACCESS.SRCE(D.61),"
1050 DATA "    DISP = OLD"
1060 DATA "//SYSUT2 DD SYSOUT=A,DCB=(BLKSIZE=80,LRECL=80,RECFM=F)"
9999  END
```

In the last example, only one job was sent. In order to send more than one job from a program, you must inform the system when there is no more to be sent for a given job. You may do this in two ways: close and reopen the job transmitter file; or execute a PRINT #n; END statement. Closing the file or printing an end-of-file tells the system that a job has been completed.

Example:

This program sends any number of jobs to a host system. The jobs are stored in ASCII files and the program asks the user what job should be sent next. The job name is also the name of the ASCII file where it is stored:

10	FILES SENDER,*	Open job transmitter file
20	DIM A\$(80)	
30	PRINT "JOB NAME";	Ask for job name
40	INPUT A\$	
45	IF A\$="STOP" THEN 150	If response is STOP, then stop
50	ASSIGN A\$,2,R	Open file where job is
60	IF R<3 THEN 90	
70	PRINT "UNABLE TO ASSIGN FILE ";A\$	If unable to open, send a message
80	GOTO 30	
90	IF END #2 THEN 130	When finished sending job, go to line 130
100	LINPUT #2;A\$	Get a line of the job
110	PRINT #1;A\$	Send it to the host system
120	GOTO 100	Loop
130	PRINT #1;END	Tell system, job is finished
140	GOTO 30	Go ask for another job
150	END	

Retrieving output from a job on your terminal is just as simple. Again you must determine what host functions are connected to which job function designators. After that you must equate an available job function designator (one you have the capability to use) with a file name.

You can now write a program to retrieve job output. All that is necessary is to read the job list or job punch file and do what you want with the lines of output received.

Example:

The following program retrieves job output and stores the results in a BASIC formatted file.

```
FILE-GETJOB,JLØ
CREATE-JOBOUT,4ØØ
LIST
```

1Ø	FILES GETJOB,JOBOUT	Opens job list file and output file
2Ø	DIM AS[134]	
3Ø	IF END #1 THEN 7Ø	When no more lines are available, stop
4Ø	LINPUT #1;AS	Ask for a line
5Ø	PRINT #2;AS	Write it to output file
6Ø	GOTO 4Ø	Loop
7Ø	PRINT " JOB SUCCESSFULLY RETRIEVED"	Inform user of completion
8Ø	END	Stop

When a host system returns lines of output to an RJE workstation, it includes information about spacing of lines. If you retrieve your job output on a line printer, the 2000 system automatically formats the lines. If you receive your job output on your terminal, you may wish to print the output on a line printer at some future time. In order to allow for this, the 2000 system includes the line printer control functions as part of each line it passes through the job list device. The first character of each line is a digit. If you convert this digit to a number and use it in a CTL function when printing to the line printer, your output will be formatted exactly as if the 2000 system were printing it. The second character of each line contains a null character. Actual print data begins with the third character of the string.

Example:

The following program retrieves job output from a host system and prints it on a line printer just as if the host system were printing it on a local line printer.

```
FILE-PRINTR,LPØ
FILE-JOBLIS,JL
LIST
```

1Ø	FILES JOBLIS,PRINTR	Open job lister and line printer
2Ø	DIM AS[134]	
3Ø	IF END #1 THEN 7Ø	Stop when no more output
4Ø	LINPUT #1;AS	Get a line of output from host
5Ø	PRINT #2;CTL(NUM(AS[1,1]));AS[3]	Tell line printer to space and print line
6Ø	GOTO 4Ø	Loop
7Ø	PRINT "JOB RETRIEVED"	Tell user all done
8Ø	END	Stop

Note: When using the 2770/2780/3780 Emulator, the second character of each line can be used to direct output to alternate devices. The first character of each line is the normal carriage control character as shown above. The second character of output returned to the program has the following assignment:

- 0 = normal device selection (printer)
- 1 = alternate device selection 1
- 2 = alternate device selection 2 (2770 only)
- 3 = alternate device selection 3 (2770 only)

IBM hosts used with an MRJE workstation emulator expect a job punch device to select one of four options when punching output. If a local paper tape punch is used to retrieve this output and the control information from IBM selects stackers 1 or 3, then the data is punched in ASCII mode format (see CTL (30)). If the control information selects stackers 2 or 4, then the data is punched in transparent mode (see CTL(33)). The system provides this control information when passing the output data to a job punch device. The mechanism is the same as with job list devices except that the first character contains only a "1" or a "2". Host systems used with the 2770/2780/3780 Emulator and CDC User 200 Emulator do not send this character since they do not support the punch function.

HOW TO GET YOUR OUTPUT

Previous discussions indicated that retrieving output is simple. There is no problem if only one host list function is connected to one job list designator. However, for IBM systems used with an MRJE workstation emulator, it is possible to have up to seven host list functions connected to any combination of real line printers or job list designators. The same applies to the multiple host punch functions. When the host is ready to return the output of a job, it generally selects the first available host list function for list output or the first available punch function for punch output. This means that if you send your job on job transmitter 3 (JT3), you are not guaranteed that you will receive the job's output on job lister 3 (JL3). You must use forms control to guarantee what host list function your output is returned to. You must also specify in your job that it must be printed (or punched) only on a device loaded with special forms.

The IBM (OS/MVT HASP) forms command is:

\$TRMx.PRn, F=f (for list output)

or

\$TRMx.PUn, F=f (for punch output)

where

x = your remote station number (assigned by the host system operator)

n = the host function number that you want the forms output returned to

f = the forms number (four digits, x or # may be substituted as a "don't care" digit)

Such a command can be specified in an RJE-RC System operator command to direct all output assigned a forms number to a particular host function. PR in the forms command is equivalent to HL (Host Lister) and PU to HP (Host Punch). For example:

RJE-RC,\$TR18.PR1,F=0005 The system operator uses this command to cause all printer output using form 0005 to be returned to Host Lister 1 (HL1) which is equivalent to PR1 in the command.

By using an "x" or "#" in place of one or more digits of the forms number, a group of form types can be routed to the same host function with a single command. For example:

RJE-RC,\$TR18.PU1,F=010# This causes all punch output using forms 0100 through 0109 to be sent to Host Punch 1 (HP1) which is equivalent to PU1 in the command.

The system operator command RJE-DA can be used to direct output on a particular host function to a particular device designator. For example:

RJE-DA,HP1>PP2 This causes output on Host Punch 1 (HP1) to be sent to Punch 2 (PP2).

RJE-DA,HL1>LP3 The system operator uses this command to cause output on Host Lister 1 (HL1) to be sent to line printer 3 (LP3).

Refer to the System Operator's manual for a description of these RJE commands as well as the other RJE commands available to the 2000 system operator.

Figure 9-4 illustrates the use of forms assignment to route output. The RJE control commands are entered by the system operator using the system console. These commands specify forms assignment; that is they specify what is to be done with output for which forms control is specified. The jobs themselves are entered through the host reader. Within the jobs there may be job control statements to specify that output be printed to a specific form. For example, a line of job 1 in the example could contain the following statement:

```
520 DATA '//SYSPRINT DD SYSOUT = (A,,0005)''
```

This data statement is read as a line of the job and specifies that output use form 0005 (PR1 to the IBM host, HL1 or LP3 to the 2000 System).

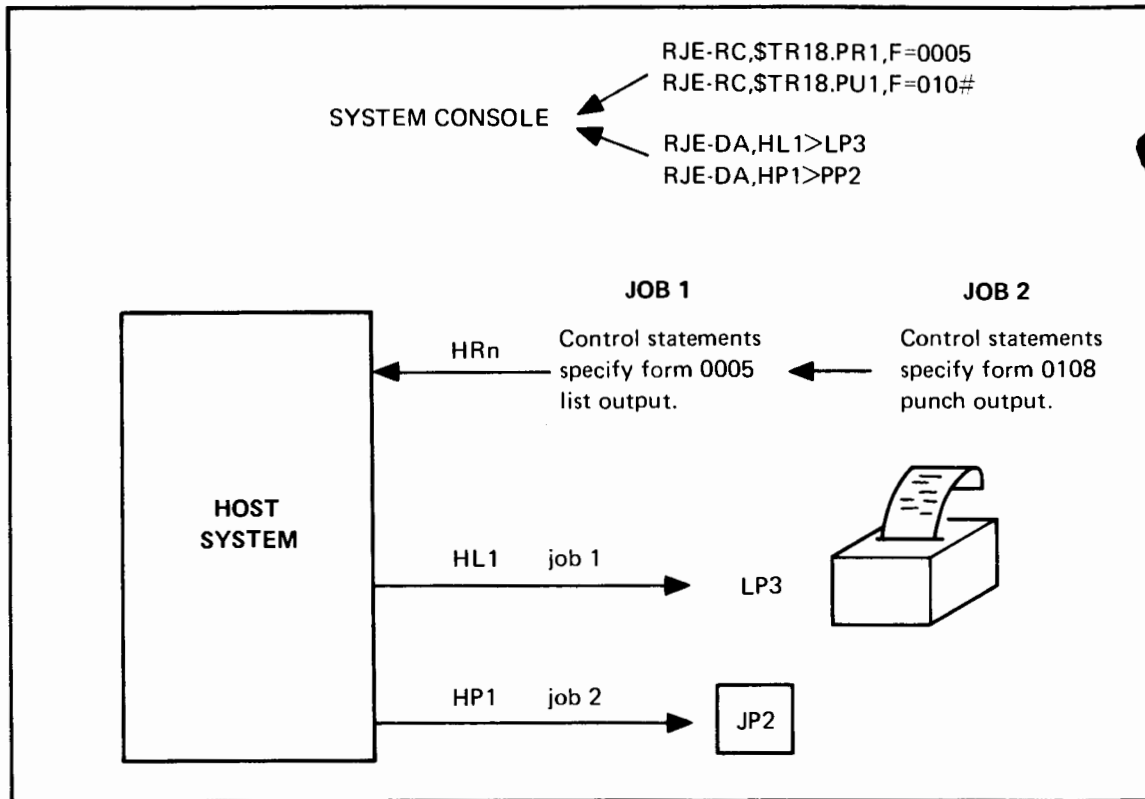


Figure 9-4. Example of Forms Assignment to Route Output
(IBM MRJE Workstation Emulation)

Note that the forms assignment performed above through the RJE-RC commands can also be transmitted as strings printed to the host message function through a job inquiry file. This method is illustrated in line 30 of the next example.

Example

The following program sets forms control in line 30 to specify a particular line printer (PR4 or HL4). The SYSPRINT card (line 522) specifies the forms control so that output is routed to the printer using form 0005. The system operator uses the system console RJE-DA command to specify that HL4 is line printer #4.

```
FILE-INQ,JI
FILE-TRANS,JT3
FILE-OUTPUT,JL4
FILE-PAYROL,DS,100
LIST
```

```
INQ = Job Inquiry
TRANS = Job Transmitter
OUTPUT = Job Lister
PAYROL = ASCII disc file
```

(Continued on next page)

10	FILES INQ,TRANS,OUTPUT,PAYROL	Open files
20	DIM A\$(134)	
30	PRINT #1;"\$TR18.PR4, F=0005"	Set forms control on Host Printer 4
40	IF TYP(0)=3 THEN 80	When out of data, go get output
50	READ A\$	Send to the host one
60	PRINT #2;A\$	line of the job
70	GOTO 40	Loop until job sent
80	ASSIGN *,2	Close transmitter to signal end of job
85	IF END #3 THEN 120	When output done, tell user
90	LINPUT #3;A\$	Get a line of output and
100	PRINT #4;A\$(3)	write it to ASCII disc file
110	GOTO 90	Go get another line of output
120	PRINT "JOB SENT AND RETRIEVED"	Tell user that we are all done
130	STOP	
500	DATA "// PAYROLL JOB (45678912), 'PROGRAMMER'"	} Actual job to be sent
510	DATA "/*PASSWORD SECRET"	
520	DATA "//PAYSTEP EXEC PGM=PAYROLL"	
522	DATA "//SYSPRINT DD SYSOUT=(A,,0005)"	
524	DATA "//SYSIN DD *"	
530	DATA "JOHN SMITH,40"	
540	DATA "MARY JONES,48"	
550	DATA "HARRY DOAKES,32"	
560	DATA "WILLIAM BLOOM,42"	
570	DATA "/*"	
9999	END	

COMMUNICATING WITH A HOST SYSTEM THROUGH A PROGRAM

Just as it is possible to send jobs and retrieve output using a program at your terminal, you may send messages and commands and receive messages from the host system with your BASIC program. Printing strings to the job inquiry file causes those strings to be sent to the host system as if they were typed on the 2000 system console. Any messages the host system sends to the 2000 system console can be read from the job message file and printed at your terminal. Note that the job inquiry and job message functions are not available on the 2770/2780/3780 Emulator.

Example:

This program allows a user to send status commands to an IBM host system and to retrieve the returned status.

FILE-INQUIR,JI
 FIL-RESPND,JM
 LIST

10	FILES INQUIR,RESPND	Open job inquiry, message file
20	DIM AS[120],BS[20]	
30	PRINT LIN(1);"WHICH JOB'S STATUS DO YOU WANT";	Prompt user for job name
40	INPUT BS	
45	IF BS="STOP" THEN 90	Stop when user types STOP
50	PRINT #1;"\$D'";BS;"'	Send \$D'name' command
60	LINPUT #2;AS	Accept response
70	PRINT AS	Print it on user terminal
80	GOTO 30	Loop
90	END	
	RUN	

```
WHICH JOB'S STATUS DO YOU WANT?DS2001
RJE $*15.11.20 JOB 617 DS2001 EXECUTING N PRIO 4

WHICH JOB'S STATUS DO YOU WANT?STOP
DONE
```

Refer to Tables 9-4, 9-5, and 9-6 for a summary of representative HASP, ASP, and EXPORT/IMPORT commands.

Whenever a line is sent to a job inquiry device, the line is also printed on the system console. This informs the system operator what a user is sending on the job inquiry function. Any messages sent by the host system are always printed on the system console.

Because any message sent to the system console is also sent to the user in control of the job message file, certain care must be exercised. In our example above, an IBM HASP Display Job command was sent and the response was then retrieved. If the operator had already sent a command which produces several lines of output, the user could have gotten one of these lines rather than the actual status requested.

CONTROLLING MULTIPLE DEVICES

When using RJE the error trapping capability (IF ERROR statement) of the HP 2000 system can be used to efficiently control multiple devices. With the error trap enabled, error 300 is returned by the system under two conditions:

1. when an input is being requested and no data is present
2. when output is requested and the IOP is currently unable to accept more data (because the buffer is full)

This feature can be used to write applications which take advantage of the multiple stream capability of the MRJE workstation.

In the example below, files SENF1 and SENF2 are BASIC formatted files containing data to be sent to the host system. Files TRAN1 and TRAN2 are job transmitters; files LIST1 and LIST2 are job listers. Files RECF1 and RECF2 are BASIC formatted files into which the data retrieved from the host through the job listers is to be placed.

The program reads SENF1 and writes to TRAN1 until error 300 is received. Then it moves to SENF2 and TRAN2, reading from SENF2 and writing to TRAN2 until the IOP accepts no further data for this stream. The program then checks the list functions (statement 310). If data is there, it is stored in the BASIC formatted file. When error 300 has been received from both job listers, the program loops back to the beginning (statement 410) and tries to send data again.

```

10  FILES SENF1,SENF2,TRAN1,TRAN2
20  FILES LIST1,LIST2,RECF1,RECF2
30  DIM AS(255),BS(255),CS(255)
40  READ #1;AS
50  READ #2;BS
100 REM
110 REM READ FROM FIRST DATA FILE, SEND OVER FIRST JOB TRANSMITTER
120 REM
130 IF ERROR THEN 170
140 PRINT #3;AS
150 READ #1;AS
160 GOTO 140
170 IF SYS(0)#300 THEN 500
200 REM
210 REM READ FROM SECOND DATA FILE, SEND OVER SECOND JOB TRANSMITTER
220 REM
230 IF ERROR THEN 270
240 PRINT #4;BS
250 READ #2;BS
260 GOTO 240
270 IF SYS(0)#300 THEN 500
300 REM
310 REM READ FROM JOB LISTERS, SAVE IN DATA FILES
320 REM
330 IF ERROR THEN 390
340 FOR L=5 TO 6
350 F=L+2
360 LINPUT #L;CS
370 PRINT #F;CS
380 GOTO 360
390 IF SYS(0)#300 THEN 500
400 NEXT L
410 GOTO 100
500 END

```

Refer to the discussion of the IF ERROR statement in this manual for more information.

REMOTE COMMAND SUMMARY

In order to use the full capabilities of your host system, you should be familiar with that system. Table 9-7 lists IBM and CDC manuals that describe the host systems available to the RJE user on a 2000 system. Tables 9-4 through 9-6 summarize commands used for remote control on these IBM and CDC systems. If you are using the 2770/2780/3780 Emulator with a HASP operating system, the remote commands listed in Table 9-4 are used. If you are using the 2770/2780/3780 Emulator with another host system, refer to the host system reference manual for a list of the remote commands.

Table 9-4. Summary of 360 HASP Remote Commands

COMMAND	COMMENTS
\$DA	Display status of active jobs
\$DF	Display information on jobs queued for printing with special forms
\$DN	Display names and status of all jobs
\$DQ	Display number of jobs in each queue type
\$CJn or \$C'name'	Cancel job with job number 'n' or job with job name 'name'
\$DJn or \$D'name'	Display status of job with job number 'n' or with job name 'name'
\$PJn or \$P'name'	Cancel specified job after completion of current activity
\$Bdevice	Backspace device
\$Cdevice	Cancel current function on device
\$Edevice	Restart current function on device
\$Fdevice	Forward space device
\$Idevice	Interrupt current function on device
\$Ndevice	Repeat current function on device
\$Pdevice	Stop the device after completion of the current function
\$Sdevice	Start the device
\$Tdevice	Set device parameters (to indicate such things as type of forms loaded in a printer, or type of print character set in use)
\$Zdevice	Halt the device immediately

Table 9-5. Summary of Some IBM ASP Remote Commands

COMMAND	COMMENTS
X,name [, message]	Specifies to ASP system the name of the support program to be scheduled for execution.
C,name	Terminates currently active support function
I,A	Displays active jobs
I,B	Displays backlogged jobs
I,D	Displays disposition of I/O devices
I,J=job name	Displays information about specific jobs
I,J=job number	Displays information about specific jobs
Z,console name,test	Sends text message to another console

Table 9-6. Summary of CDC EXPORT/IMPORT Remote Commands

COMMAND	COMMENTS
ON	Turns equipment logically on. Card readers are initially on, and output devices are initially off.
OFF	Turns equipment logically off.
DEFINE	Specifies the various attributes of an output device (print train, forms code, etc.).
WAIT	Temporarily halts reading or printing.
GO	Resumes operation after a WAIT command.
BSP	Backspaces a print file a specified number of sectors.
END	Terminates current operations on the specified equipment.
REP	Used to specify the number of additional copies of the file in process to be printed.
REW	Rewinds the print file in progress and turns the equipment logically off.
RTN	Returns a print file to the appropriate queue with its present priority or a newly specified priority.
SUP	Suppresses the spacing of a print file, so that the remainder of the file is single spaced.
DIVERT	Allows the user to divert output files of a remote job to the central site or another terminal.
DROP	Allows a user to drop a job which is currently in execution.
EVICT	Allows a user to eliminate a job from the input and/or output queues.
KILL	Allows a user to kill a job which is currently in execution.
PRIOR	Allows a user to change the priority of a file in the output queue.
REVERT	Cancels the effect of a DIVERT
H	Displays the contents of the terminal's input and output queues.

Table 9-7. Some Useful Host System Manuals

HOST SYSTEM	MANUAL	DESCRIPTION
IBM	HASP Operator's Guide GC27-6993	Description of HASP remote operator commands
IBM	ASP Operator's Guide GH20-1289	Description of ASP remote operator commands
CDC	CDC 200 User Terminal 82128000	Description of terminal operating procedures
CDC	Computer Systems Reference Manual 60100000	General system reference
CDC	INTERCOM Reference Manual 60307100	General system reference
CDC	SCOPE 3.4 Reference Manual 60307200	General system reference
CDC	KRONOS General Information Manual 60407100	General system reference
IBM	2770 Data Communications System GA27-3013	Description of terminal operating procedures
IBM	2780 Data Transmission Terminal GA27-3005	Description of terminal operating procedures
IBM	3780 Data Communications Terminal GA27-3063	Description of terminal operating procedures

2000 TO 2000 COMMUNICATIONS

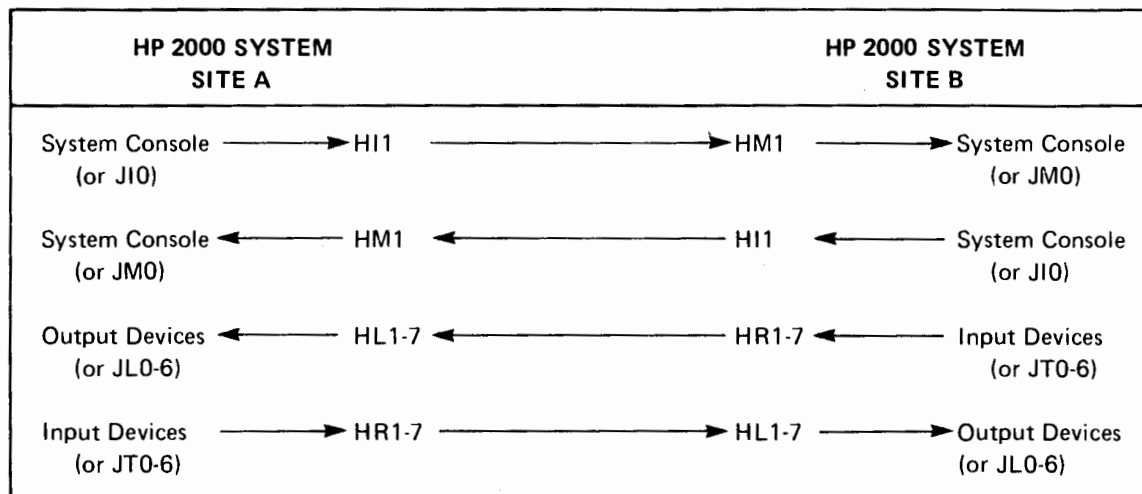
Communication between two HP 2000 systems uses the resources of RJE. In many ways it is very similar to the process of Remote Job Entry, except that only data, not jobs, can be transmitted between 2000 systems. As with RJE, messages can be sent and received through the system console or through the job inquiry and message functions. Data can be transmitted from one 2000 system to another through the job transmitter functions and received through the job lister functions.

When a local HP 2000 system uses RJE to communicate with a remote 2000 system, the remote system can be considered the host. However, when the host returns data to the local system, then that system acts as host. Each 2000 system may use seven reader functions for accepting input and seven lister functions for producing output, an inquiry function for sending messages, and a message function for receiving messages. Except for the lack of a punch function, this is like the IBM MRJE workstation.

As with RJE communication, input can be placed directly in a card reader equated to a host reader or data may be communicated programmatically using an ASCII file equated to a JT (Job Transmitter) function. Output may be received directly on a host lister or programmatically through an ASCII file equated to a JL (Job Lister) function.

Table 9-8 illustrates the communications link between two HP 2000 systems. The JI, JM, JL, and JT alternatives to the console or devices are job function designators described earlier under the heading "How Does RJE Work?".

Table 9-8. Communication Between Local and Remote 2000 Host Functions



Note that 2000 to 2000 communication does not support a host punch function.

TRANSMITTING DATA BETWEEN SYSTEMS

When a local 2000 system enters data through a local host reader function (card reader, paper tape reader, reader/punch/interpreter), that data is transmitted to the matching host function of the remote 2000 system. The data may then be displayed on the physical device associated with the host function (line printer, paper tape punch, reader/punch/interpreter) or it can be read through a job lister function associated with the host function. Similarly, data transmitted from a host reader function at a remote 2000 site is received at the local site through the corresponding host lister function and displayed on the associated output device or read from the job lister function.

In 2000 to 2000 communication, anything sent through a job inquiry function is printed on the system console at the receiving end. Messages sent through the job inquiry function can be read by the job message function just as with Remote Job Entry.

When sending lines on a job transmitter, standard line printer control codes (CTL functions) may be used to specify the carriage control characters to be used by the receiving system. Host lister functions use these control codes to format output to any line printers associated with the host lister.

Example

The following program reads a deck of cards from a card reader at site A and sends the data to site B to be listed on the line printer. The line printer control functions are used to print 25 double-spaced lines per page.

Site A Program

FILE-CARD, CR
FILE-OUT, JTO

```

10  FILES CARD,OUT
20  DIM A$(80)
30  IF END #1 THEN 100
40  PRINT #2;CTL(1)
50  FOR I=1 TO 25
60  LINPUT #1;A$
70  PRINT #2;CTL(4);A$
80  NEXT I
90  GOTO 40
100 END

```

Set printer at site B to top of form
Set up FOR loop to read 25 cards
read card
send card image to site B, double-spaced

At site A, JT0 is assigned to HR1 thus the card images with control functions are transmitted through the Host Reader 1 to site B. At site B, a line printer is assigned to Host Lister 1, the host function that receives the card images.

Note that the CTL functions as used in this example of 2000 to 2000 communication could not be used to control the line printer at CDC or IBM host stations since those systems merely expect card image data.

TRANSMITTING DATA BETWEEN PROGRAMS

2000 to 2000 communication can be used to transmit data from a BASIC program executing at one site to a BASIC program executing at another site. If the data is saved on files and is not to be printed on the line printer or other output device, the receiving program should strip the first two characters of each line so that the line printer control characters are not transferred.

Example:

The following programs illustrate the transfer of a disc file named DATA containing 80 characters per record from a system at site A to another system at site B.

Site A Program

FILE-OUT, JT0, 40

```

10 FILES DATA, OUT
20 DIM A$(80)
30 IF END #1 THEN 70
40 LINPUT #1; A$
50 PRINT #2; A$
60 GOTO 40
70 END

```

Stop at end of data in DATA
 Read a line from DATA into A\$
 Send the line to site B through file OUT
 Loop back for next line

Site B Program

FILE-IN, JL0, 41

```

10 FILES IN, DATA
20 DIM A$(82)
30 IF END #1 THEN 70
40 LINPUT #1; A$
50 PRINT #2; A$(3)
60 GOTO 40
70 END

```

extra word for control characters

Stop when no more data in IN (job lister)
 Read line from IN into A\$
 Strip control character and null character;
 save line in DATA

2780 TO 2780 COMMUNICATION

This facility allows communication between an HP 2000 system emulating a 2770, 2780 or 3780 terminal, and one of the following:

- a 2770, 2780 or 3780 terminal (with the same configuration as the Emulator)
- another HP 2000 system running the 2770/2780/3780 Emulator
- an HP 3000 system running the 2780/3780 Emulator
- any other computer system emulating a 2770, 2780 or 3780 terminal (with the same configurations as the Emulator)

Refer to Table 9-9 for the exact terminal configurations provided by the 2770/2780/3780 Emulator. In all four types of communications listed above, the terminal configurations being emulated on the HP 2000 system must match the terminal configurations being emulated on the other system.

2780 to 2780 Communication is similar to the process of Remote Job Entry except that only data, not jobs, can be transmitted. Data can be transmitted from one system to another through the job transmitter functions and received through the job lister functions.

As with RJE communication, input can be placed directly in a card reader equated to a host reader or data may be communicated programmatically using an ASCII file equated to a JT (Job Transmitter) function. Output may be received directly on a host lister or programmatically through an ASCII file equated to a JL (Job Lister) function. Except for the fact that only one reader function and one lister function are available, the capability is similar to the IBM MRJE workstation.

Table 9-9. Terminal Configurations of 2770/2780/3780 Emulator

2770	Buffer expansion (256 characters each), fifteen response retries, EBCDIC code, EBCDIC transparency, automatic answering, component selection (note that this is only useable by BASIC programs).
2780	Automatic answering, multiple record transmission (basic 400 character buffer), EBCDIC code, EBCDIC transparency, 144 character print lines (note that HP printers only support up to 132 character print lines), extended ENQ retry, component selection (note that this is only useable by BASIC programs).
3780	EBCDIC code, EBCDIC transparency, automatic answering/disconnect, space compression/expansion, extended ENQ retry, 144 character print lines (note that HP printers only support up to 132 character print lines), component selection (note that this is only useable by BASIC programs). The 3780 always runs as if the space compression/expansion switch is on.

This section contains a definition of what 2000 user commands are and how they are used. General terms used to discuss the commands are defined early in the section. Following the term definitions a complete description of each user command is given. (Commands used by the System Operator are defined in the operator's manual.)

WHAT IS A COMMAND?

Commands instruct the system to perform control or utility functions such as storing and listing programs or logging on and off the system. Commands differ from the statements used to write a program in the BASIC language in that a command instructs the system to perform some action immediately, while a statement is an instruction to perform an action only when the program is executed or run. A statement is always preceded by a statement number; a command is not.

Any command can be entered once the logging on procedure has been successfully completed. Commands can be abbreviated to the first three characters. (Embedded blanks are ignored.) If a command is misspelled, the system will return three question marks indicating that it did not recognize the command. If a command is not received properly, the system will print "TRANSMISSION ERROR. REENTER". You should retype the command. Following entry of each command, *return* must be pressed to signal that command entry is complete.

Most commands have parameters to further define command operation. For instance, the LIST command causes a display of your current program. It may have parameters to specify that only part of the program is to be printed, or that the program is to be listed to a device other than your terminal. If parameters are used, a hyphen or the "*OUT= file name*" construct (refer to specific commands for a discussion of this form) is used to separate the command from its parameters. If multiple parameters are used, they are usually separated by commas.

Example:

```
LIS-20,100
```

This would list the program beginning with statement 20 (or the first statement with a number greater than 20) and continuing through statement number 100.

TERMS USED IN THIS SECTION

The following pages contain definitions of terms that will be used in the command descriptions given later in this section. These terms will be used to define in detail the operation of each of the commands.

ASCII File

An *ASCII file* is an area on the disc or a *non-sharable device* which can be accessed through file LINPUT, READ, and PRINT statements. ASCII files are created with the FILE command. The FILE command equates a *file name* with either an area on the disc (ASCII disc file) or with a *non-sharable device*. ASCII files can contain only string data. Note that a number can be represented by a string (e.g., "123.45" represents the number 123.45).

Examples:

FILE-DFIL, DS, 40	(creates an ASCII disc file of 40 <i>blocks</i>)
FIL-PUNCH, PP	(creates an ASCII file equated to the paper tape punch)
FIL-READER, CR, 40	(creates an ASCII file equated to the card reader)

Use of ASCII disc files is granted to every user on the system (provided that their *idcode* has been granted the use of sufficient disc space). The system operator grants use of *non-sharable devices* to an *idcode* through the NEWID or CHANGEID commands described in the 2000 System Operator's Manual (22687-90005).

You can access ASCII files as if they were your terminal. Each *record* of an ASCII file is read and/or written as if it were a line on a terminal.

Any special characteristics of devices used as ASCII files are described in the following paragraphs. The function CTL (refer to Section XI) provides control over devices when used as ASCII files.

LINE PRINTER. When using a line printer as an ASCII file, opening the file causes a page eject. Closing the file or using END in a print list has no effect on the line printer. The message LPn-DEVICE NOT READY is issued when the device is not ready; the device may usually be readied by pressing the on-line button, clearing a paper jam, or inserting new paper. If a line is interrupted by a system power failure, it is automatically reprinted. CTL functions may be used to skip lines or to set the printer at top-of-form (refer to Section XI).

PAPER TAPE READER. When using the paper tape reader as an ASCII file, the first read causes a search for the first non-null character. Null characters on paper tape are represented by no punches except the small sprocket hole. A frame of null characters is also known as a "feed frame". Closing the file has no physical effect. A record consisting of 60 adjacent feed frames is interpreted as an end-of-file. The following error conditions may occur during a read:

PRn-DEVICE NOT READY — Device was opened but is not ready because there is no tape in reader, too much leader (300 inches), device is turned off, or tape has been read out of reader. Correct the condition and continue.

PRn-READ/WRITE FAILURE — A system power failure occurred while the device was in use, or a parity error was detected; also returned if attempt is made to write data to the reader. The only corrective action is to restart the program.

Paper tape conventions are the same as for the terminal unless a CTL function specifies that the tape be read in "transparent mode" where each frame is treated as a valid 8-bit character. Other CTL functions allow reading with odd or even parity checks. (Refer to CTL function in Section XI for details).

PAPER TAPE PUNCH. When using the paper tape punch as an ASCII file, opening and closing the file punches leading and trailing feed frames respectively. Using END in a print list causes 20 feed frames to be punched. Records are terminated with X-OFF, a carriage return and line feed unless a CTL function specifies otherwise. The message PPn-DEVICE NOT READY is issued when the tape supply is low or the device is not turned on when opened. Any line whose punching is interrupted by a system power failure is terminated with a control-X, carriage return and line feed (to allow any later reading to ignore it) and then repunched.

CARD READER. When using the card reader as an ASCII file, opening and closing the file causes no special processing. An end-of-file is signalled in two ways: A card beginning with a double colon (: :) is read, or if the operator has pressed the EOF button on an HP 2892 type reader, an end-of-file is assumed after the last card is picked from the hopper. When cards are read, any trailing blanks are stripped from the images read to the program unless a CTL function has specified that column binary be read. CTL functions may be issued in a prior PRINT statement to specify that ASCII, column binary, or EBCDIC code is read; the default is ASCII. (Refer to CTL function description, Section XI).

The following error messages may occur during a read from the card reader:

CRn-DEVICE NOT READY — Device is off-line, there is a pick failure, the hopper is empty, the stacker is full, and so forth. Operator must perform appropriate action to ready the reader.

CRn-DEVICE ERROR — Physical failure of the unit while reading the last card. Replace the last card read and ready the device. This error could occur if the power fails when a card is in motion.

CRn-ATTENTION NEEDED — An illegal hole pattern was detected in the last read card. The operator and user must establish a correction procedure and then the operator restarts with the AWAKE command.

CRn-READ/WRITE FAILURE — Attempt made to write to the card reader.

A system power failure causes one of the first two messages above to be issued; recovery is by the procedure appropriate to the message.

CARD READER/PUNCH/INTERPRETER. This read/write device is used to read cards or to punch and interpret (print) cards. Since the read operation does not eject the card into a stacker, but leaves it in the visible wait station, a card may be read, the data manipulated, and the same card may then be punched and/or printed. The next read request ejects the last card read into a stacker. Particular hoppers and stackers may be specified with CTL functions (refer to Section XI). The following error conditions can occur during a read operation:

RPn-DEVICE NOT READY — Reader/punch/interpreter is off-line, the power is off, there was a pick failure, the hopper is empty, or the stacker is full. After the situation is corrected, the operation will continue automatically.

RPn-ATTENTION NEEDED — Invalid punch code read; correct last card, if desired, and replace in correct hopper, then operator should use AWAKE command to restart the device.

An end-of-file condition occurs when a double colon (: :) card is read.

When a request is made to punch or print a card, the card is automatically fed from a previously selected hopper if no card is in the visible wait station. The following error condition may occur when punching or printing:

RPn-DEVICE NOT READY — The interpretation and correction is the same as for read.

If system power fails, a card being read should be replaced in the appropriate hopper. If a card was being punched, it should be discarded as a new card will be punched.

The CTL functions used to select the operation, to select hoppers and/or stackers for this device are described in Section XI.

LINK TERMINAL. This read/write device is used to enter and display data and to activate indicator lights. In addition it can receive HPIB message and route them to devices connected to its local bus. Opening a link terminal as an ASCII file causes a status check to occur and initializes all units connected to the terminal. When the file is closed, the link terminal is returned to its initial state. The following error conditions may occur during a read or write operation.

LTn — DEVICE NOT READY — link terminal is off-line or the power is turned off. Once operator corrects situation, operation continues automatically.

LTn — ATTENTION NEEDED — Invalid hardware status was detected during the last read operation. The operator restarts device with the AWAKE command, and notifies user that device is now ready.

LTn — DEVICE ERROR — Physical device failure was detected during a read operation. Operator must ready the device and retype last input line. If the cause of the error condition was a power failure, the Restart Key may be used to restore the state of the terminal indicator lights. A system power failure also causes this message to print during an input operation.

CTL functions enable you to control the indicator lights, the special functions keys, and optional units connected to the link terminal.

MAGNETIC TAPE. When using a magnetic tape unit as an ASCII file, opening it causes a check to see if it is on-line and at load point. When the file is closed, an end-of-file mark is written if the last request issued was a write; in all cases, the tape is rewound. It is also unloaded unless a rewind is in progress when the file is closed. Including END in a print list writes an end-of-file mark and then backspaces over it leaving the tape positioned at the end of the current file.

Empty lines, or null records, are simulated by writing a record containing two ASCII blanks. Records with an odd number of characters are padded with an ASCII blank when written; when read this extra blank is not detected, but is read as part of the data. Reading an end-of-file mark causes a fatal error unless the IF END statement specifies action to be taken at the end of file. Reading or writing past an end-of-tape mark or reading after a write with no intervening operation also causes the end-of-file action to be taken.

The following error messages may be issued when reading from or writing to a magnetic tape ASCII file:

MTn-ATTENTION NEEDED — Device is not on-line or not at load point, or there was a hardware malfunction or an attempt to read blank tape. Operator must set the tape on-line or position the tape to load point to correct first two situations; following an attempt to read blank tape, the tape is backspaced until it is positioned prior to the last record preceding the blank tape.

MTn-READ/WRITE FAILURE — Attempt to write on file when no write ring is present, inability to read a record after three tries, the inability to write a record after six tries, or a system power failure causes this message. The only corrective action is to insert the write ring if necessary and start again. Note that the system attempts to rewind and unload any tape belonging to an open file after recovery from a system power failure; however, it is unable to do so if power was lost on the magnetic tape unit itself; in this case, the system operator must manually rewind and unload the tape.

CTL functions permit you to position the tape at a beginning of file or at a tape mark (refer to Section XI, CTL function).

BASIC Formatted File

A *BASIC formatted file* is an area on the disc which can be accessed through file READ and PRINT statements. BASIC formatted files are created through use of the CREATE command or the CREATE statement (refer to Section XI for a discussion of the CREATE statement). The CREATE command associates a *file name* with an area on the disc. BASIC formatted files can contain both string and numeric data. In addition, these files contain internal information about the type of each data item (string or number) in the file..

Examples:

CREATE-BFIL, 20 (creates a BASIC formatted file of 20 records)
 CRE-BIGFIL, 2000 (creates a BASIC formatted file of 2000 records)
 CRE-SREC, 100, 64 (creates a BASIC formatted file of 100 records of 64 words each)

Use of BASIC formatted files is granted to all users whose *idcode* has been granted disc space through the system operator's NEWID and/or CHANGEID commands.

BLOCK

A *block* is the physical unit of storage on the disc. All of the discs on a system are organized as a contiguous set of *blocks*. Each block contains 256 words (a number occupies two words; a character occupies one half of a word). When a *BASIC formatted file* is created, you specify how many *records* long it should be. Each record will be stored in one *block*. When an *ASCII disc file* is created, you specify how many *blocks* long it should be. One difference between an *ASCII disc file* and a *BASIC formatted file* is that the former can store more than one *record* per *block*.

DEVICE DESIGNATOR

A *device designator* is a two or three character mnemonic which specifies either a *non-sharable device* or a *job function designator*. There are two types of *device designators*, general and specific. When an ASCII non-disc file is created with the FILE command, it is equated to a *device designator*, either specific or general.

A *general device designator* specifies *non-sharable devices* and *job function designators* by class. The numbers in parentheses indicate the range of the digit that appended to the *general device designator*, indicates a *specific device designator*.

Commands

NON-SHARABLE DEVICE

magnetic tape
line printer
card reader
paper tape reader
paper tape punch
reader/punch/interpreter
link terminal

GENERAL DEVICE DESIGNATOR

MT (0-3)
LP (0-6)
CR (0-6)
PR (0-6)
PP (0-6)
RP (0-6)
LT (0-30)

JOB FUNCTION DESIGNATOR

job transmitter
job lister
job punch
job inquiry
job message

GENERAL DEVICE DESIGNATOR

JT (0-6)
JL (0-6)
JP (0-6)
JI (0)
JM (0)

A *specific device designator* names a specific *non-sharable device* or *job function designator*. *Specific device designators* are formed by appending a digit, 0 through 6, to the *general device designator*. A list of all *specific device designators* available to your *idcode* can be obtained with the **DEVICE** command.

Examples:

LP0, CR3, JM1, PP6, JT4, etc.

END-OF-FILE MARK (EOF)

An *end-of-file mark* is a physical or logical indication stored in a file which indicates the current end of data. When created, each record of a disc file has an *end-of-file mark* written into it as the first item in the record.

FILE LENGTH

The *file length* indicates the size of a disc file. *File length* is always specified in terms of the number of *blocks* the file occupies on the disc. The length of any file must be less than 32,767 *blocks*. When creating a disc file, the *file length* is also limited by the amount of disc space allocated to an *idcode* and the amount of unused disc space remaining in the system.

FILE NAME

A *file name* is a symbol composed of 1 to 6 letters or digits. It is used to identify a specific file in a *library*. In some cases the name can be qualified. A preceding dollar sign (\$) indicates the file is to be found in the *system library*. A preceding asterisk (*) indicates the file is to be found in the *group library*. Following the name with a period (.) and *idcode* indicates the file is to be found in the *library* of that *idcode*.

Examples:

FILE12, 3A, MYFILE, \$SFILE, *GFIL, HERS.A312

FULL DUPLEX

Full duplex is a term used to denote a method of character transmission from your terminal to the system. *Full duplex* means that when a character is typed on your terminal, it is sent to the system, but is not printed (or displayed) at your terminal. When the system receives the character, it immediately sends it back to your terminal causing it to be printed (or displayed). Also, see Half Duplex.

GENERAL DEVICE DESIGNATOR

(See *device designator*)

GROUP LIBRARY

A *group library* is an area for the storage of programs and files. There are a possible 260 *group libraries* on any system. Your *group library* has the same first two characters as your *idcode*, for example A300 is the *group library* for accounts A300 through A399, F700 for F700 through F799. The last two digits of a *group library idcode* are always zero. You may access your *group library* by prefacing a file or program name with an asterisk.

Examples:

A300, Z400, F900

HALF DUPLEX

Half duplex denotes a method of character transmission from your terminal to the system. *Half duplex* means that when a character is typed on your terminal, it is sent to the system and is also printed (or displayed) at your terminal. When the system is in *half duplex* mode, the system does not send the character back to your terminal. If you operate a *half duplex* terminal on a port set for full duplex, every character is displayed twice (once by the terminal and once by the system). To set your port to *half-duplex* mode, type ECHO-OFF.

Example:

Assume a *half-duplex* terminal is being used, but port is set for *full duplex*:

HHELL--AA112233,PPAASS	characters are repeated
8-15-75 PORT 23	HELLO program displays this line
EECCHH--OOFFFF	after this command characters will not be repeated

IDCODE

An *idcode* is a letter followed by three digits that serves to identify users of a system. The letter may be A through Z; the three digits may be 000 through 999.

Examples:

A123, B400, A000, Z999, etc.

Each user of the system has an *idcode* which is used to log onto the system. Each *idcode* can have a password, some number of minutes allowed for system connection, a number of disc *blocks* allowed for storage of programs and files, the capability to access non-sharable devices, and capabilities for Program/File Access (PFA), File Create/Purge (FCP), and Multiple Write Access (MWA).

JOB FUNCTION DESIGNATOR

A *job function designator* is a *device designator* which refers to one of the logical data paths between the 2000 system and a host system or another 2000 system. There are five *job function designators*: JM, JI, JP, JL, and JT (job message, job inquiry, job punch, job lister, and job transmitter). *Job function designators* are used as *ASCII files* for data communications.

LIBRARY

A *library* is an area of storage for programs and files. *Libraries* are specified by the *idcode* which 'owns' the entries. There are three levels of *libraries* on the system: your personal *library*, *group libraries* (one of which is your *group library*), and the *system library*. Files and programs which reside in *libraries* other than your own may be referenced in several ways. Your *group library* can be referenced by preceding names with an asterisk (*). The *system library* can be referenced by preceding names with a dollar sign (\$). Other *libraries* can be accessed by following a name with a period (.) and the *idcode* which 'owns' the *library*.

Examples:

MYPROG, \$SYSPRG, *GRPROG, OWNER.F105

LIBRARY NAME

A *file name* or *program name*.

NON-SHARABLE DEVICE

A *non-sharable device* is any peripheral device attached to the system which can only be utilized by one user at a time. *Non-sharable devices* include magnetic tape drives, line printers, card readers, card reader/punch/interpreters, paper tape readers, and paper tape punches. In addition, all of the *job function designators* are *non-sharable devices*.

OUT = FILE NAME

The **OUT = file name** construct can be used with certain commands. In commands followed by parameters, it replaces the hyphen. In commands without parameters, it simply follows the command. It causes the output of the particular command that is normally sent to your terminal to be sent to the *ASCII file* specified by *file name*. User commands that can use the **OUT = file name** construct are RUN, EXECUTE, LIST, PUNCH, CATALOG, GROUP, LIBRARY, and DEVICE. The three special system master commands used from A000 (DUMP, DIRECTORY, and REPORT) are also allowed use of the **OUT = file name** construct.

Examples:

```
RUN*OUT=LPRTR*50
CAT*OUT=PUNCH*
DIRECTORY*OUT=MAGT*L402
```

PROGRAM NAME

A *program name* is a symbol composed of one to six letters or digits. It is used to identify a specific program in a *library*.

Examples:

PROG, PR1234, TABLEP

PROGRAM REFERENCE

A *program reference* is an optionally qualified (\$, *, .idcode) *program name* used to identify a specific program in a *library*. A *program reference* may refer to a *group library* program by using an asterisk (*) prefix. A *program reference* may refer to a *system library* program by using a dollar sign (\$) prefix. Following a name with a period (.) and *idcode* indicates that the named program is to be found in the library of that *idcode*.

Examples:

\$PROG, *GRPROG, MYPROG, 5HP298.A123

RECORD

A *record* is the logical unit of storage on each device. All files on the system are made up of one or more *records*. The length of a *record* is its *record length*.

RECORD LENGTH

Record length is the number of words contained in a *record*. BASIC formatted files have *record lengths* of 64 to 256 words and each record occupies one disc *block*. ASCII disc files have *record lengths* of from 1 to 255 words and several *records* may occupy one disc *block*. *Non-sharable devices* have *record lengths* of from 1 word to the maximum number of words allowed by the system. A DEVICE command lists the maximum *record length* available for *non-sharable devices*.

SPECIFIC DEVICE DESIGNATOR

(See device designator.)

STATEMENT NUMBER

A *statement number* is an integer in the range 1 to 9999 inclusive. Each statement in a program is preceded by a unique *statement number*. It is used to indicate the proper order of the statement relative to other statements in the program.

SYSTEM LIBRARY

The *system library* is an area for the storage of programs and files. There is one system library per system and its *idcode* is A000. The system library is the personal library of *idcode* A000, but entries within it may be referenced by other *idcodes* by prefacing a name with a dollar sign (\$). The *system library* is also the *group library* for *idcodes* A000 through A099.

WORK SPACE

Your *work space* is an area where your current program is stored. When statements are entered at your terminal, they become part of your *work space*. When you GET a program it is brought into your *work space*. A *work space* may have a name associated with it through use of the NAME or GET command. When a program is saved, that name becomes the name of the program in your *library*. The SCRATCH command clears your *work space* and its name.

COMMAND DESCRIPTIONS

The following pages describe each of the user commands available on the system. Each command description gives the general form of the command, a description of command operation, and any optional parameters.

The description for each command is followed by one or more examples. The descriptions use the terms defined in the previous pages. Whenever terms appear they are printed in *italics*. Terms that are defined locally within the command description itself are emphasized by printing them in ***bold italics***. Optional parameters are enclosed in square brackets ([optional parameters]).

Refer to table 10-1 for a list of the available user commands.

Table 10-1. User Commands

APPEND	HELLO	PROTECT
BYE	KEY	PUNCH
CATALOG	LENGTH	PURGE
CREATE	LIBRARY	RENUMBER
CSAVE	LIST	RUN
DELETE	LOAD	SAVE
DEVICE	LOCK	SCRATCH
ECHO	MESSAGE	SWA
EXECUTE	MWA	TAPE
FILE	NAME	TIME
GET	PAUSE	UNRESTRICT
GROUP	PRIVATE	

APPEND Command

General Form:

APPEND-program reference

The APPEND command retrieves the named program from the appropriate *library* and appends it to the current contents of your *work space*. The name of your *work space* is unchanged.

The lowest *statement number* of the appended program must be greater than the highest *statement number* of the current program. You can append any program in your library except programs that have been saved with the CSAVE command or that contain the COM statement.

You can append unrestricted or protected programs from your *group library* or the *system library* or from *libraries* with the PFA (Program/File Access) capability. If a protected program is appended from another *library*, the resulting program in your *work space* will also be protected. Programs that are locked or private can be appended only by their owners.

Examples:

```
APP-$PUBLIC
APP-HIS.B904
APP-MINE
APP-*GROUP
```

BYE Command

General Form:

BYE

The BYE command is used to log off of the system.

Entry of the BYE command ends the current session. The system will respond by printing the number of minutes that you have used in the current session and will also update the total time used by your *idcode*. If you are using a modem, telephone connection is broken.

Example:

```
BYE
0009 MINUTES OF TERMINAL TIME
```


CATALOG, GROUP, and LIBRARY Commands

General Forms:

CATALOG [*-library name*]
GROUP [*-library name*]
LIBRARY [*-library name*]

The CATALOG, GROUP, and LIBRARY commands are used to obtain alphabetical listings of programs and files stored on the system.

If the *library name* parameter is used, the listing begins with the first program or file with a name that is the same or follows the *library name* in alphabetic or numeric order.

The CATALOG command lists the names of all programs and files stored in your *library*. The GROUP command lists the names of non-private programs and files stored in your *group library*. The LIBRARY command lists the names of non-private programs and files in the *system library*.

The hyphen in the command may be replaced by the **OUT = file name** construct.

Programs, *BASIC formatted files*, and *ASCII disc files* are listed in the following format:

Program or file name; *program or file descriptor; access restriction; length; record*

All other *ASCII files* are listed in the following format:

File name; A; *access restriction; specific or general device designator; record*

A *program or file descriptor* is one of the following characters:

A — *ASCII file*
F — *BASIC formatted file with SWA (Single Write Access)*
M — *BASIC formatted file with MWA (Multiple Write Access)*
C — *CSAVED program*
blank — *SAVED program*

An *access restriction* is one of the following characters:

U — *UNRESTRICTED*
P — *PROTECTED*
L — *LOCKED*
blank — *PRIVATE*

Length is the number of *blocks* required to store the program or file. *Record* is blank for programs and for *BASIC formatted files* having a *record length* of 256 words. For other *BASIC formatted files* and *ASCII files*, *record* is the *record length* in words.



Examples:

CAT*OUT=DISC* Send library listing to the disc.
 CAT*OUT=LPR*T List on the line printer all programs or files in your library
 beginning with T through Z.
 GRO List all accessible programs and files in your group library.

LIB-P											
NAME	LENGTH	RECORD	NAME	LENGTH	RECORD	NAME	LENGTH	RECORD	NAME	LENGTH	RECORD
P1	AL	10	63	P2	AL	10	63	PAY	CU		2
PP0	AL	PP0	64	PRIN	U			PTTY	CU		2
PZ075	U			PZ125	U			PZ325	U		15
RATES	CU			SCOOP	CU			SCR	FL		100
SNLIST	FL			STATS	CU			T	FL		20
TESTER	U			TRADER	L			TRADES	L		29
TSP001	CL			TSP002	CL			TSP003	CL		25
TSP004	CL			TSP005	CL			TSP006	CL		23
TSP007	CL			TSP008	CL			TSP009	CL		8
TSP010	CL			TSP06A	CL			TSP06B	CL		10
TSPHSP	L			TSPMSG	FL			TSPU01	L		17
TSPU02	L			UWBSPI	CU			XCO RUN	CU		26

CREATE Command

General Form:

CREATE-file name, file length [, record length]

The CREATE command builds a *BASIC formatted file* on disc. *End-of-file (EOF)* marks are written into each record. One disc *block per record* is used regardless of the *record length*. The *file length* may be limited by the system configuration and the disc space allotted to your *idcode*. The file is created in the LOCKED state with SWA (Single Write Access) access. You can create files in your own *library* only, so the *file name* cannot be qualified.

Examples:

CRE-FILEA, 48
 CRE-MYFILE, 123, 128

CSAVE Command

(Refer to SAVE)

DELETE Command

General Form:

DELETE-beginning statement number [ending statement number]

The DELETE command erases all statements in your *work space* between and including the specified statements.

Commands

If both *statement numbers* are the same then only the specified statement is deleted. If the ending *statement number* is given, it must not be less than the beginning *statement number*. If no ending *statement number* is given, the delete operation continues through the end of the program. DEL-1 has the same effect as the SCRATCH command except that your *work space* retains the *program name*.

Examples:

```
DEL-27
DEL-27,50
DEL-27,27
```

DEVICE Command

General Form:

DEVICE

The DEVICE command lists the *specific device designators* for the *non-sharable devices* that are available to your *idcode* in the following format:

DEVICE DESIGNATOR	MAXIMUM RECORD LENGTH	STATUS
----------------------	--------------------------	--------

Maximum record length is the largest *record length*, in words, specifiable for the device in a FILE command. The status field may be blank, indicating that the device is available for use; may contain the word BUSY indicating that the device is being used by another user; or may contain the characters N/A indicating that the system operator has removed the device from the system or assigned it for exclusive use by another *idcode* or RJE.

The **OUT = file name** construct may be used with the DEVICE command.

Example:

```
DEV
```

DEVICE DESIGNATOR	MAXIMUM RECORD SIZE	STATUS
CR0	80	
CR1	80	
LP0	66	BUSY
LP1	66	
JM0	60	
JP0	41	
JL0	67	
JT0	66	
PR0	64	N/A
RP0	160	
LT0	40	
LT1	20	

ECHO Command

General Form:

ECHO-ON

or

ECHO-OFF

The ECHO command causes the system to echo or not echo characters received from your terminal and allows use of *half-duplex* terminals.

Users with a *half-duplex* terminal should type the ECHO-OFF command immediately after logging on.

ECHO-ON returns a port to the *full-duplex* mode wherein characters are echoed.

Examples:

```
ECH-OFF
ECH-ON
```

EXECUTE Command

General Form:

EXECUTE-*program reference*

The EXECUTE command clears your *work space* of any previous program, brings the specified program into your *work space*, and begins executing the program.

Execution always begins at the first program statement. When the program terminates, or any chained-to programs terminate, the *work space* is automatically cleared. You can always execute your own programs. Programs that are unrestricted, locked, or protected and are saved in the *system library*, your *group library*, or in any *library* having the PFA (Program/File Access) capability can also be executed. You cannot execute private programs saved in libraries other than your own.

The **OUT= file name** construct can be used in place of the hyphen in the EXECUTE command.

Examples:

```
EXE-MYPROG
EXE-HERPRG.B456
EXE*OUT=LPR**GROUP
EXE-$SYSTEM
```

FILE Command

General Form:

FILE-file name, device designator [, record length]

or

FILE-file name, DS, file length [, record length]

The FILE command builds an *ASCII file*. (A general description of ASCII file usage is contained in Section V).

The first general form builds an *ASCII file* by merely equating *file name* to the *specific or general device designator* for a *non-sharable device*; no disc space is allocated in this case. The *device designator* must be configured into the system and your *idcode* must be endowed with the capability to access the general device class. Additionally, the device (if *device designator* is specific) or a member of the device class (if *device designator* is general) must be currently available to your *idcode*. (The system operator can dynamically make specific devices unavailable or assign them exclusively to RJE or to a particular *idcode*). *Record length*, if specified, is an integer between 1 and a device dependent maximum determined by the system configuration. This number is the largest *record* you will permit the file to contain, i.e., the maximum *record length*. A list of the specific devices available to your *idcode*, with their maximum *record length* and current status, can be obtained with the DEVICE command.

If a *general device designator* is specified and the optional *record length* parameter is supplied, there must exist at least one device of that type, with a maximum *record length* greater than or equal to that specified, which is currently available to your *idcode*.

ASCII files built with the FILE command are not opened until they are referenced in a program. At that time, if a maximum *record length* was not specified for the *ASCII file*, the file will assume the maximum *record length* of the *non-sharable device*. If the file is associated with a *general device designator* the system will assign the lowest numbered specific device of that type that is available to your *idcode* and has a sufficient maximum *record length*.

The second general form builds an ASCII disc file. *File length* specifies the number of disc *blocks* to be allocated and may be limited by the system configuration and the amount of disc space allotted to your *idcode*. *End-of-file* (EOF) marks are written over the entire allocated disc area. *Record length* sets an upper limit on the number of words per record and must be an integer between 1 and 255 inclusive; 63 words is the default. Records exceeding this maximum are truncated on the right, shorter records occupy only the space actually required (ASCII data is packed 2 bytes per word and records must begin on word boundaries, therefore the byte following an odd-length record is unused). A word indicating the *record length* precedes each record. Records cannot cross *block* boundaries; if the number of words remaining in a *block* is less than *record length* plus 1, the entire record is written in the succeeding *block*.

All ASCII files are created in locked state with Single Write Access (SWA). (Refer to Section VIII).

Examples:

```
FIL-LIST,LP,60
FILE-FASTLP,LP2
FILE-MYFILE,DS,100
FIL-CDMAST,DS,5000,40
```

GET Command

General Form:

GET-program reference

The GET command clears your *work space* of any previous program and brings the specified program into your *work space*.

Programs that are private or locked can only be accessed with the GET command by their owner. Programs that are protected or unrestricted residing in a *library* with PFA (Program/File Access) capability or in the *system library* or your *group library* can also be accessed with the GET command.

Examples:

```
GET-MYPROG
GET-HISPRG.C119
GET-$REC
GET-*ACT
```

GROUP Command

(Refer to CATALOG)

HELLO Command

General Form:

HELLO-*idcode,password* [, *terminal type*]

The HELLO command is used to log on to the system. Your *idcode* and *password* are assigned by the system operator. The *terminal type* tells the system what type of terminal you are using.

The *idcode* is the account number assigned to you by the system operator; it is always a single letter followed by three digits.

Your *password* consists of from 0 to 6 printing or non-printing characters. Entering your password properly validates access to the system. Use of non-printing characters allows a degree of security in that other users cannot see your password.

When selecting a *password* you should be aware that the following non-printing characters cause undesirable effects on certain terminals:

CHARACTER	EFFECT
J ^C (linefeed) @ ^C (null) H ^C (backspace) M ^C (carriage return) S ^C (X-OFF) X ^C (line delete)	Stripped by the system (all terminals)
ESCAPE	Initiates control sequence on HP 2640, HP 2644, and TermiNet terminals
E ^C (ENQ)	Before logging in, an E ^C typed on an HP 2640 or HP 2644 causes an F ^C (ACK) to be sent; after logging in with terminal type 1 or 2, system strips F ^C entered immediately after E ^C .
G ^C (bell)	Rings a bell

Specify your *terminal type* as a numeric digit between 0 and 8. If you are not sure of your terminal type, you may consult Appendix D where a list of representative terminals is provided for each type, or you may ask your system manager. Failure to specify the correct *terminal type* can result in a loss of data. If *terminal type* is omitted, the system assumes type 0.

When the HELLO command has been entered, the system validates your *idcode*, *password*, and so forth and then executes a BASIC program called \$HELLO. While this program is executing, the BREAK key is disabled just as if you had executed the BRK(0) function. This prevents accidental termination of the HELLO program. The HELLO program may re-enable the BREAK key to allow you to terminate.

Example:

```
HEL-B123,PSWRD,1
```

KEY Command

General Form:

KEY

The KEY command tells the system that the following input will be from your terminal keyboard. It is used only after a TAPE (paper tape input) operation is complete. It causes error messages suppressed by the TAPE command to be output to the terminal.

Any valid command has the same effect as KEY except that commands substituted for KEY in this manner are not executed if diagnostic messages (indicating syntax errors in BASIC statements) were generated during tape input.

Example:

```
KEY
```

LENGTH Command

General Form:

LENGTH

The LENGTH command prints the number of words in the program currently in your *work space*, followed by the number of blocks needed to save the program (slightly more space may be needed to CSAVE the program). The total disc space used and allocated to your *idcode* is also printed.

Example:

```
LEN
00151 WORDS=01 RECORDS. 00201 RECORDS USED OF 65000 PERMITTED.
```

LIBRARY Command

(Refer to CATALOG)

LIST Command

General Form:

LIST

or

LIST-P

or

LIST-[beginning statement number] [, ending statement number] [, P]

The LIST command produces a listing of statements in your *work space*, in *statement number* order. Beginning and/or ending *statement numbers* can be specified to obtain a partial listing. If your *work space* has a *program name*, that *program name* precedes the listing.

The letter "P" may be used to produce page formatted output. Fifty-six lines of output are generated per "page unit" together with blank lines to space the listing for cutting into 11-inch sheets for binding or filing.

Listings may be terminated with the BREAK key. PROTECTED programs obtained from other *libraries* cannot be listed.

The **OUT = file name** construct may be used in place of the hyphen. When **OUT = file name** is used the P parameter is ignored.

Examples:

```
LIS
LIS*OUT=PRTR*
LIS*OUT=HOLD*100,200
LIS-22
LIS-P
LIS-,400
LIS-50,P
LIS-100,400
```

LOAD Command

General Form:

LOAD-*file name*

The *file name* must specify an ASCII file. The LOAD command loads the contents of the ASCII file into the user's *work space* as a program. It assumes the file contains a group of BASIC statements.

This command is similar to the TAPE command that reads BASIC statements from paper tape into the user's *work space*. LOAD reads one line (*record*) at a time from the specified ASCII file. If the line consists of a syntactically valid BASIC program statement, the statement is merged with the current contents of the work space. Lines that do not begin with a *statement number* are ignored without comment; a line that begins with a *statement number* but is not syntactically valid is rejected and an error message is generated that is passed to the user when the command completes execution.

LOAD is completed when an end-of-file is read, when the BREAK key is pressed, or when a fatal file error (READ/WRITE FAILURE) is generated by the ASCII file device.

The *file name* may be fully qualified if security restrictions allow.

Examples

The following example creates ASCII file CREADR as a card reader, clears the user work area, and subsequently reads the program source image from the card reader to the user work area:

```
FILE-CREADR,CRO
SCR
LOAD-CREADR
```

The next example reads a program from a disc ASCII file in account A123:

```
SCR
LOAD-PRGM.A123
```

LOCK Command

General Form:

LOCK-*library name*

The LOCK command places the named program or file in the locked state. Refer to Section VIII for a description of program and file states.

Note that if the LOCK command is successful, the return variable value is set to zero and the pointer is reset to the beginning of the record.

Examples:

```
LOC-MYPROG
LOC-MYFILE
```

MESSAGE Command

General Form:

MESSAGE-*character string*

The MESSAGE command sends a *character string* preceded by your port number to the system operator.

Commands

The *character string* can be up to 68 characters long. Longer strings are truncated on the right. All printing characters (including quotes, commas, blanks, etc.) are accepted; except for BELL (control-G), non-printing characters are stripped.

If the system operator's message storage area is full, the message:

CONSOLE BUSY

will be printed on your terminal, indicating that the message has not been sent and should be entered again.

Example:

MES-PLEASE MOUNT TAPE GT476 ON DRIVE 3

MWA Command

General Form:

MWA-*file name*

The MWA command places the file specified by *file name* in the MWA (multiple write access) state. Your *idcode* must be endowed with the MWA capability. See the ASSIGN and FILES statement discussions in Section XI for a description of multiple write access.

Example:

MWA-MYFILE

NAME Command

General Form:

NAME [-*program name*]

The NAME command assigns a name to the program currently in your *work space*.

If NAME is entered with no *program name*, no name is assigned to your *work space*. If a name was previously assigned, it is deleted.

Examples:

NAME-PROGR1
NAM-ADDER
NAM-MYPROG
NAM

PAUSE Command

General Form:

PAUSE-*time limit*

The PAUSE command causes an executing program to pause for the number of seconds specified in *time limit*. PAUSE only has effect if executed through the SYSTEM statement (refer to SYSTEM statement description in Section XI). If no program is executing PAUSE has no effect.

After the number of seconds specified in *time limit* have elapsed, the program is automatically restarted. The system operator may also restart a paused program by typing *AWAKE-port number*.

The *time limit* is specified as an integer between 1 and 32767. If a number smaller than 1 or greater than 32767 is specified, or if the parameter is incorrectly formatted, a 1 is returned in the SYSTEM statement *return variable*.

If PAUSE is successful, a zero is returned to the SYSTEM statement *return variable*. Note that the PAUSE command may pause for two or three seconds longer than the specified *time limit*, but will never pause for less time than specified.

Examples

```
10 SYSTEM N,"PAU-600"
```

PRIVATE Command

General Form:

PRIVATE-*library name*

The PRIVATE command places the named program or file in the private state. Refer to Section VIII for a discussion of program and file states.

Examples:

```
PRI-MYPROG  
PRI-MYFILE
```

PROTECT Command

General Form:

PROTECT-*library name*

The PROTECT command places the named program or file in the protected state. Refer to Section VIII for a discussion of program and file states.

Examples:

PRO-MYPROG
PRO-MYFILE

PUNCH Command

General Form:

PUNCH[P]

or

PUNCH-[*beginning statement number*][, *ending statement number*][, P]

The PUNCH command punches the current program onto paper tape if your terminal has a paper tape punch. The program is punched in *statement number* order. Starting and/or ending *statement numbers* can be specified to obtain a tape containing a partial program. In addition, the *program name* and leading and trailing feed holes are also punched. The program is listed on your terminal as it is punched. If your terminal does not have a punch, only a listing will be generated.

The terminal punch should be turned on after the punch command is typed, but before the trailing carriage return is typed.

An X-OFF, carriage return, and line feed are added at the end of each line to enable other BASIC programs to read the paper tape as data or to allow reentry of the program tape. The X-OFF character gives the system control over reading the tape by turning off the paper tape reader while the system processes the line just read. The system sends an X-ON character to the reader when it is ready to read the next line. The X-ON character instructs the reader to continue reading. (See the TAPE command.)

The letter "P" may be used to produce page formatted output. Fifty-six lines of output are punched per "page unit" together with *linefeeds* to space the output for later listing. This allows for cutting listings into 11-inch sheets for binding or filing. The "P" option will not otherwise affect the tape being punched.

Punching may be terminated with the BREAK key. Protected programs obtained from other *libraries* cannot be punched or listed.

The `*OUT= file name*` form may be used to divert the output to an ASCII file. When `*OUT= file name*` is used the "P" parameter is ignored. If the punching is directed to an ASCII file, that file must be a PP device.

Examples:

```
PUN
PUN*OUT=TAPE*
PUT*OUT=OUTBUF*100,200
PUN-22,55
PUN-P
PUN-,400,P
PUN-50,P
PUN-,400
```

PURGE Command

General Form:

PURGE-*library name*

The PURGE command deletes the specified program or file from your *library*. It will not affect *work space*. A purged program or file is recoverable only if another copy exists.

A file may not be purged while it is being accessed by another user. The PURGE command can be used to dissociate *non-sharable devices* from the name used in a previous FILE command. If the file is an *ASCII disc file* or *BASIC formatted file*, the file space that it occupied is returned to the system.

Examples:

```
PUR-PROG12
PUR-FILE09
```

RENUMBER Command

General Form:

RENUMBER

or

RENUMBER-statement number

or

RENUMBER-statement number, Interval

or

RENUMBER-statement number, Interval, beginning statement number

or

RENUMBER-statement number, Interval, beginning statement number, ending statement number

The **RENUMBER** command is used to renumber statements in your *work space*. The initial *statement number* specifies what the new *statement number* of the first affected statement should be. *Interval* is the difference between successive *statement numbers*. Starting and ending *statement numbers* refer to the old *statement numbers* at which the renumbering is to begin and end. *Statement numbers* referenced in BASIC language statements are automatically replaced with the appropriate new number unless they reference non-existent statements in which case they remain unchanged.

If the initial *statement number* is not given, the first *statement number* of the renumbered program will be 10. If no *interval* is specified then new numbers will be in increments of 10. If the beginning *statement number* is not given, the renumbering will begin with the first statement of the program. If the ending *statement number* is not given the renumbering will continue to the end of the program. If both beginning and ending *statement numbers* are absent then the entire program will be renumbered.

If all parameters are omitted then the entire program is renumbered with the first statement numbered 10, at intervals of 10. **RENUMBER** cannot be used to change the order of statements. If any command parameter is omitted then all of the parameters following it must also be omitted.

Examples:

```
REN
REN-100
REN-10,1
REN-20,50,100
REN-10,10,50,100
```

RUN Command

General Form:

RUN

or

RUN-statement number

The RUN command starts execution of the current program in your *work space* (do not confuse it with the EXECUTE command). Execution normally starts with the first statement in a program. When a *statement number* is given, execution begins at the specified *statement number* or the next highest *statement number* if that specified statement does not exist.

Note that when the RUN-statement number form of the command is used, all statements before the specified statement will be skipped. Variables defined in skipped statements will be undefined and cannot be referenced until they are defined in an assignment, INPUT, ENTER, READ, or LINPUT statement. FILES, DIM and COM statements are executed before any other statements regardless of where they appear in a program. They are always executed.

A running program may be terminated with the BREAK key. If you were allowed to GET a program from another *idcode*, you are allowed to RUN it.

The *OUT= *file name** construct may be used with the RUN command.

SAVE and CSAVE Commands

General Form:

SAVE

or

CSAVE

The SAVE command is used to save a copy of the current program. A copy of the program is transferred from your *work space* to your *library*. A program must have a name assigned to it before it can be saved (see the NAME command).

Example:

SAV

Commands

The CSAVE command is also used to save a copy of your current *work space*. The version saved by the CSAVE command will begin execution slightly faster than a saved program. This is especially important with large programs that do a lot of chaining.

Example:

CSA

A protected program from another user's *library* may not be SAVED nor CSAVED.

SCRATCH Command

General Form:

SCRATCH

The SCRATCH command deletes the entire current program including the *program name* from your *work space*. Scratched programs are lost unless they have been saved elsewhere.

Example:

SCR

SWA Command

General Form:

SWA-*filename*

The SWA (Single Write Access) command removes the named file from the MWA (Multiple Write Access) state. The file may now only be written to by the first user to access it while that user has the file opened. The single write access state is the default state for files. The SWA command places any MWA file in SWA state whether or not your account has MWA capability.

Example:

SWA-MYFILE

TAPE Command

General Form:

TAPE

The TAPE command tells the system that the following input (a group of BASIC statements) is from the terminal's paper tape reader.

TAPE suppresses any diagnostic messages which are generated by input errors, as well as the automatic *linefeed* after *return*. The KEY command causes any diagnostic messages that have been generated to be output to your terminal, ending the TAPE mode.

The tape reader on your terminal must be turned on before typing the *return* after the TAPE command, since the system will respond immediately to this command with an X-ON character which turns on the tape reader and initiates reading. Refer to the PUNCH command.

The paper tape must contain an X-OFF, carriage return, and linefeed at the end of each line. Also, your terminal paper tape reader must turn off when it reads an X-OFF character and turn back on when an X-ON is received from the system.

Example:

```
TAP
```

TIME Command

General Form:

TIME

The TIME command responds with one line containing your *idcode*, port number, time used since log-on, total time used for the *idcode*, and maximum time permitted for that *idcode*. All times are expressed in minutes.

Example:

```
TIM
```

```
A000 ON PORT # 05 FOR 00025 MIN. 00125 MIN USED OF 65000 PERMITTED.
```

UNRESTRICT Command

General Form:

UNRESTRICT-*library name*

The UNRESTRICT command places the named program or file in the unrestricted state. Refer to Section VIII for a description of program and file states.

Examples:

UNR-MYPROG
UNR-MYFILE

BASIC LANGUAGE REFERENCE

SECTION

XI



INTRODUCTION

This section contains descriptions of the statements and functions used in programming on the 2000 system. A separate heading is used for each statement and function. The headings are arranged in alphabetical order. Following each heading will be either the description for the function or statement or a reference to where the description occurs. Some descriptions have been grouped together functionally for purposes of discussion. Note that statement numbers, while required for all statements are not included in the general form descriptions.

BASIC LANGUAGE TERMS

Each description is made up of the statement or function format, explanatory text, and one or more examples. The format is made up of the statement or function together with any required and optional parameters or arguments. The parameters and arguments are described using a set of global BASIC language constructs or terms. These terms are described in the following paragraphs. When these terms are used in descriptions they will be printed in *italics*. Terms which apply only to specific statements or functions are defined within the statement description. These terms are printed in *bold italics*.

ARRAY

An *array* is an ordered collection of numbers referenced either simultaneously by the associated *array name* or individually by qualifying the *array name*. The second form of reference is called a *subscripted variable* and the individual value specified is called an *array element*. For instance, D(3,8), S(46), and X(101,3) are all *subscripted variables* referring to values that are *array elements*.

An *array* can be one-dimensional, organized as a column of elements (individually referenced by a *subscripted variable* with a single subscript, the value of which specifies the element's position within the column). An *array* can also be two-dimensional, organized as one or more rows of one or more columns of elements (individually referenced by a *subscripted variable* with two subscripts, the value of the first specifying the row and the value of the second specifying the column which contains the element). Within a program all references to elements of a given *array* must be consistent with that *array's* dimensionality. Two-dimensional *arrays* are rectangular; that is, all rows have the same number of columns.

Array A		4.5
(one-dimensional)	=	3
		102
		73

Array B		59	3	96
(two-dimensional)	=	1	5	32
		12	95	2

where:

A(1) = 4.5
A(2) = 3
A(3) = 102
A(4) = 73

where:

B(1,1) = 59	B(1,2) = 3
B(1,3) = 96	B(2,1) = 1
B(2,2) = 5	B(2,3) = 32
B(3,1) = 12	B(3,2) = 95
B(3,3) = 2	

Each *array* has a size, defined as the number of elements it contains. The size of a one-dimensional *array* is simply the number of elements in its single column. The size of a two-dimensional *array* is the product of the number of rows and columns. The size of an *array* can be specified in a DIM statement or COM statement; as few as 1 or as many as 5000 elements can be specified in this way.

Example:

```
DIM M(32), F(20,10), X(500)
```

Array M has 32 elements; Array F has 20 rows and 10 columns or 200 elements; Array X has 500 elements.

If not specified, the system will assign a size of 10 to a one-dimensional *array* or a size of 100 (10 rows by 10 columns) to a two dimensional *array*. Each *array element* occupies 2 words of user work space. Although the definitions of the language allow each *array* in a program to have as many as 5000 elements, the capacity of the 2000 system imposes lower limits in most cases. The available user work space can contain slightly more than 5000 *array* elements. Thus one *array* of maximum size is possible. However, this space must be shared with the program itself, other *arrays*, *numeric simple variables*, *string simple variables*, buffer space for files, etc. Even a program using only a single *array* and few or none of the above items must be restricted to a small number of statements in order to leave enough space for 5000 elements.

In general an *array* has no attributes beyond those discussed above. A program can treat a one-dimensional *array* as the mathematical entity referred to as a column vector and can treat a two-dimensional *array* as the mathematical entity referred to as a matrix. The MAT statement of BASIC includes several forms which operate on *arrays* according to the rules of vector and matrix arithmetic. Most forms of the MAT statement perform functions on *arrays* which are useful for non-matrix purposes as well.

ARRAY ELEMENT

(Refer to *array*).

ARRAY NAME

An *array name* is a single alphabetic character (A through Z) used to reference a collection of numbers called an *array*. Within a program a given *array name* will always refer to the same unique *array*. However, the form of an *array name* is also one of the allowable forms of a *numeric simple variable*. It is permissible for the same symbol (letter) to be used both as an *array name* and as a *numeric simple variable* within the same program. In each appearance of the symbol, its context determines which of the two names it represents.

Example:

```
110 PRINT K(3,2) , P(201) , K
```

In this example, there is no confusion between the *numeric simple variable* K and the *array name* K because the *array* is subscripted.

CHARACTER

A *character* is any member of the ASCII character set (refer to Appendix A). Most terminals do not provide a means to enter all of the 128 characters defined by ASCII. In addition the system strips the following characters from paper tape reader or terminal input: null, control-H, linefeed, carriage return, X-OFF, control-X, and rubout. Although within the 2000 system each character occupies an 8-bit field (one half of a word), the most significant bit of each character received from a terminal or paper tape reader is forced to 0, since this bit is used only for parity-checking rather than character differentiation. However, any of the 256 possible 8-bit configurations can be generated internally if needed. String assignments preserve all 8 bits of each character and string comparisons treat each of the 256 different character values as unique.

CONSTANT

A *constant* is either a *numeric constant*, optionally preceded by a plus sign (+) or minus sign (-), or a *literal string*.

Example:

-3.2, 1.59 E3, "ABC", and "ST" '32 '65 are all constants

DESTINATION STRING

A *destination string* is a *string variable* to which a value is assigned in a BASIC language statement. BASIC includes a number of constructs which assign a value to a *string variable*. For example, in each of the following statements A\$ is a *destination string*:

```
10 LET A$="ABC"
20 INPUT A$
40 CONVERT N TO A$(3)
40 SYSTEM A$(3,80),"TIM"
```

If the *destination string* is a *string simple variable*, its entire current value is replaced by the value assigned to it and its *logical length* is set to the length of the value assigned to it. If the length of the value assigned to the *destination string* exceeds the physical length of the *string simple variable*, an error results.

Example:

Assume A\$ currently has the value "ABC" (and, thus, a logical length of 3), and a physical length of 255 characters.

```
22 A$="EFGH"           A$ now has the value EFGH and the logical length of 4
```

If the *destination string* is a *string variable* with a single *substring designator* then replacement by the assigned value begins at the character position specified by the *substring designator*. The initial portion preceding the *substring designator* retains the current value. The new *logical length* of the *destination string* is set to the specified character position plus the length of the assigned value -1 . An error results if the new *logical length* exceeds the *physical length* of the *destination string*. An error also results if the specified character position exceeds the current *logical length* $+1$.

Examples:

Assume A\$ has the value "EFGH" and a logical length of 4.

23 A\$(3)="AB"	A\$ now has the value EFAB and the same <i>logical length</i> , 4
24 A\$(5)="C"	A\$ now has the value EFABC and <i>logical length</i> , 5.
25 A\$(7)="X"	An error results since position 6 is undefined.

If the *destination string* is a *string variable* with a double *substring designator*, replacement begins with the first specified character position and continues through the second specified character position. The current value outside this range is retained. The number of characters replaced is always equal to the second value $-$ first value $+ 1$. If the assigned value contains more characters than needed, only its initial portion is used. If it contains too few characters then blanks are added on the right. In the special case of a null *substring designator* where the second value $-$ first value $+ 1 = 0$, no replacement occurs.

If the last character position in the *substring designator* is greater than the current *logical length*, it becomes the new *logical length*. If this new length exceeds the *physical length* of the *destination string*, an error results. An error also results if the number of characters to be replaced is less than zero (the second value $-$ first value $+ 1$ is less than zero).

Examples:

Assume A\$ = "ABCD" and its physical length is 255.

31 A\$(2,3)="CB"	A\$ now has the value ACBD.
32 A\$(4,8)="EF"	A\$ now has the value ACBEF ^^^ and a logical length 8. Note that 3 blanks are appended on the right.
33 A\$(1,1)="GH"	A\$ now has the value GCBEF ^^^. Note that the second character assigned was truncated.
34 A\$(1,0)="I"	A\$ is unchanged since the substring defines a null string, that is, a string whose length is zero.
35 A\$(10,11)="J"	An error results since the logical length of A\$ is 8 and the <i>substring designator</i> leaves position 9 undefined.
36 A\$(2,0)="XX"	An error results since the number of characters replaced is less than zero.

FILE NAME

A *file name* is a symbol composed of 1 to 6 letters and/or digits. It is used to identify a specific file in a library. In some contexts the name can be optionally qualified (\$, *, or idcode). A preceding dollar-sign (\$) indicates that the named file is to be found in the system library. A preceding asterisk (*) indicates that the named file is to be found in the group library. Following the name with a period (.) and account number indicates that the named file is to be found in the library of that account number.

Example:

Valid File Names	Invalid File Names
2 FILE	\$\$STAR (\$ is not a letter or digit)
FILE	FILE332 (more than 6 characters)
A	FILE #1 (# is not a letter or digit)
TSPMSG	
YRPG1.C302	
*GFILE	(An initial * indicates GFILE is in a group library)
\$LUGHA	(An initial \$ indicates LUGHA is in the system library)

FILE NUMBER

A *file number* is a *numeric expression* which evaluates to a number associated with a file in an executing BASIC program. The *numeric expression* is evaluated and rounded (if necessary) to an integer. The result corresponds to a number associated with an open file by the FILES or ASSIGN statement, or if the *file number* is zero, to the user's terminal.

FUNCTION

A function is a name followed by one or more arguments in parentheses that, when referenced, produces a result. The 2000 system defines the following numeric valued functions plus two string valued functions. In the following list, a *numeric expression* is represented by x and a *source string* by s.

Numeric-Valued Functions

ABS(x)	returns absolute value of x.
ATN(x)	returns arctangent of x.
BRK(x)	depending on value of x, enables or disables BREAK key, or returns current status of BREAK key.
COS(x)	returns cosine of x.
CTL(x)	performs control functions for ASCII file devices; the particular function depends on the value of x.
EXP(x)	returns value e^x .
INT(x)	returns integer part of expression x.
ITM(x)	returns number of data items from beginning of the current record of BASIC formatted file x.

LEN(s)	returns the current <i>logical length</i> of source string s.
LIN(x)	depending on value of x, optionally performs carriage return and zero or more line feeds in print operation.
LOG(x)	returns natural logarithm, i.e. $\log_e x$.
NUM(s)	returns numeric ASCII code value of first character in string s.
POS(s ₁ ,s ₂)	returns character position in string s ₁ where s ₂ (if a substring of s ₁) starts.
REC(x)	returns current record number in BASIC formatted file x.
RND(x)	generates pseudo random number based on value of x.
SGN(x)	returns value indicating sign of x.
SIN(x)	returns sine of x.
SPA(x)	spaces x character positions in print operation.
SQR(x)	returns square root of x.
SYS(x)	depending on value of x, returns error information trapped by IF ERROR, BREAK key operation if break disabled, or terminal type.
TAB(x)	moves to print position x in print operation.
TAN(x)	returns tangent of x.
TIM(x)	depending on value of x, returns current minute, hour, day, year, or second.
TYP(x)	depending on value of x, returns data type, end-of-data, end-of-file, end-of-record in a file or a DATA statement.

String-Valued Functions

CHR\$(x)	returns character corresponding to numeric ASCII code x.
UPSS\$(s)	returns upper-case equivalent to ASCII characters in string s.

Each of these functions is described fully later in this section; the descriptions are in alphabetic order by name. In addition to these pre-defined system functions, user defined functions can be defined in a DEF statement. All such functions have names beginning with FN followed by a letter (A through Z) and must be numeric-valued with a single numeric expression as their argument.

FUNCTION REFERENCE

A *function reference* is a function name followed by the actual argument within parentheses. It is used to request evaluation of the named function. A numeric value results from evaluating a numeric-valued function, a string value from a string-valued function.

Examples:

```
9000 A3 = ATN(33)
9010 P(45) = POS(A$, "11")
9020 PRINT NUM(B$(31,31)), FNA(X)
9030 CONVERT FND (Z+10) TO D$
```

LITERAL STRING

A *literal string* is a textual representation of a string value within a BASIC program. The usual form consists of a pair of double quote marks enclosing an ordered sequence of characters (blank characters are significant). However, some characters cannot be represented in this fashion since most terminals do not use the full ASCII character set, the system uses certain characters for special control purposes and strips them from paper tape reader or terminal input, and the double quote mark is used as the delimiter. By convention an apostrophe followed by a decimal integer (in the range 0 to 255 inclusive) is interpreted as specifying the equivalent internal 8-bit character. Any character can be specified with this numeric equivalent convention (most can only be specified in this manner). A general *literal string* can be composed of any mixture of quoted character sequences and individual characters represented with the numeric equivalent convention as long as no two quoted character strings are adjacent. Note that an apostrophe appearing within a quoted character string is simply the character apostrophe.

A *literal string* can contain as few as 0 characters or as many as 255. The former is called the null string and is represented by two adjacent double quote marks (" "). The maximum number of characters accepted in a line generated by a terminal can be as few as 80 or as many as 256 and can vary from terminal to terminal on the same 2000 system. In practice a *literal string* in a BASIC statement will always be accepted if the entire statement does not exceed 80 characters (all blanks are included in the count but the terminating carriage return is not). The length of a *literal string* is the number of characters it contains (not the number of characters used to represent it) and the value is the ordered collection of these characters.

Examples:

```
"ABC"
"AB" '67 '68
"F" '32 "G"
'10 '13 '72
"HJKL"
" "
"NINE TAILOR'S NEEDLES"
```

LOGICAL LENGTH

The *logical length* is the number of characters in the current string value of a *string simple variable*.

Example:

```
5 DIM A$(30) – the physical length of A$ is 30 characters.
If A$ = "BEGIN PROGRAM" then the logical length of A$ is 13.
```

LOGICAL SIZE

The *logical size* is the current working size of an *array*. The MAT statement permits changing the size of an *array* during execution. The *logical size* can range between 1 and the *physical size* of the *array*. The *logical size* of a one-dimensional *array* is simply its current working size. The *logical size* of a two-dimensional *array* is the product of its current row size and its current column size.

Example:

```
10 DIM A(32), M(6,8), Z(3000)
20 MAT A = CON(16)
```

The *physical sizes* of arrays A, M, and Z are 32, 48, and 3000 respectively. Initially, the *logical size* is set equal to the *physical size*. Statement 20 sets the *logical size* of array A to 16.

NEW DIMENSIONS

New dimensions can be either the number of rows in a one dimensional array enclosed in parentheses or the number of rows and columns in a two dimensional array separated by a comma and enclosed in parentheses. The number of rows or the number of columns may be *numeric expressions* which are evaluated and rounded to integers. The optional *new dimensions* allows you to respecify the number of rows and columns of an array. These *new dimensions* must be within the limits specified in the original DIM or COM statement or within the default limits set by the system for the array.

In addition the total number of elements (rows for a one dimensional array or rows \times columns for a two dimensional array) in the newly dimensioned array cannot exceed the number of elements dimensioned for in the original array.

Example:

```
10 DIM A(10,15), B(20,20)
:
:
80 MAT A = CON(10,10)
90 MAT READ B(15,15)
```

Arrays A and B are originally dimensioned 10×15 and 20×20 respectively. Statement 80 changes the working size of array A to 10×10 and sets each element to a one. Statement 90 changes the working size of array B so that only 225 elements will be read from a DATA statement.

NUMBER

A *number* is an approximation to a real number. It is used as the value represented by a *numeric constant*, referenced by a *numeric variable*, or resulting from evaluation of a *numeric expression*. Real numbers are represented with approximately six (seven in a portion of the range) decimal digits of precision. The ranges of real numbers in the 2000 system are -10^{38} through -10^{-38} , 0, and 10^{-38} through 10^{38} .

NUMERIC CONSTANT

A *numeric constant* is a textual representation of a number. Its value is the closest approximation possible to the real number specified by the form of the *numeric constant*. The simplest forms are an integer (an optional + or - sign followed by a sequence of one or more digits) and an integer followed by a decimal point (.), optionally followed by an integer. Either of these forms can be followed by an exponent (the letter E followed by a

one or two-digit integer in the range of 0 to 38 inclusive, which is optionally preceded by a plus sign (+) or minus sign (-). An exponent has the value of 10 raised to the positive or negative power indicated by the following integer. The number represented by the complete form is the value of the integer or fraction multiplied by the value of the exponent, if present. Note that you can specify '.', '—.', '0.0', etc., to represent zero.

Examples:

Numeric Constant	Value
19.796	19.796
-6.768 E 10	-67680000000
123	123
31.45678E-3	.0314568

NUMERIC EXPRESSION

A *numeric expression* is a combination of operators and *primaries* which can be evaluated to a number. The general form of a *numeric expression* is too complex to explain all at once. The discussion will first describe a simpler form and then proceed to generalize it. A simple description of numeric expressions as they are commonly used is included in Section II. The most common form is the 'arithmetic expression', which closely resembles the simple algebraic expression of mathematics. An arithmetic expression can be a sequence of one or more 'sums', where each sum (except the last or only one) is syntactically separated from its successor by either MIN or MAX. During evaluation these symbols are interpreted as binary operators. Starting with the first sum, each MIN or MAX compares the value of the preceding portion of the *numeric expression* with the value of the following sum and selects the minimum or maximum respectively as its result. The result of the last MIN or MAX is the value of the entire *arithmetic expression*.

A sum is a sequence of one or more 'terms', where each term (except the last or only one) is syntactically separated from its successor by either a '+' or '-'. During evaluation these symbols are interpreted as binary operators specifying addition or subtraction respectively. A term is a sequence of one or more 'factors', where each factor (except the last or only one) is syntactically separated from its successor by either a '*' or '/'. During evaluation these symbols are interpreted as binary operators specifying multiplication or division respectively. Each factor (including the first or only one) can by option have a preceding '+' or '-'. During evaluation these symbols in this context are interpreted as unary operators. The unary '-' specifies negation of the value of its following factor; the unary '+' exists only for syntactic symmetry and has no effect on evaluation. A factor is a sequence of one or more *primaries*, where each *primary* (except the last or only one) is syntactically separated from its successor by either '^' or '**'. These symbols are alternative representations of the same binary operator, called the exponentiation operator. During evaluation they specify that the value of the *primary* following the operator is used as a power and the value of the preceding portion of the factor is used as a base (e.g., A**B is equivalent to the mathematical notation A^B).

The use of '+' and '-' as both binary and unary operators is traditional in mathematics. Unfortunately it can create confusion in some cases. Consider the following three legal arithmetic expressions:

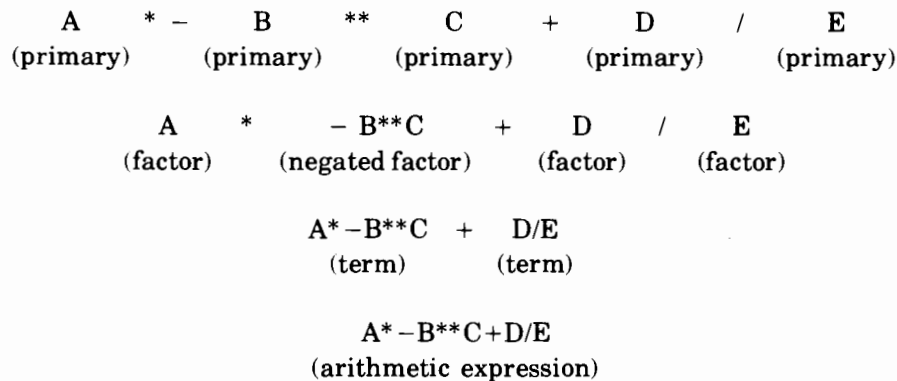
term - factor

term - + factor

term + - factor

By definition any term except the last one must be followed by a '+' or '-', while a factor is only optionally preceded by a unary '+' or unary '-'. Thus the proper interpretation of the first case is as 'term - term', where the second term is a simple factor. The second case is also 'term - term', where the second term is a factor preceded by a unary '+' and hence is equivalent to the first case. The third case is 'term + term', where the second term is a factor preceded by a unary '-'. Under the rules of mathematics, subtraction of a number is equivalent to addition of its negative value. The result is that all three of the above cases will evaluate to the same value.

The following diagram can be read from top to bottom as an illustration of how *primaries* and operators combine to form an arithmetic expression, or it can be read from bottom to top as an illustration of how the definition of an arithmetic expression is expanded into its components.



In any arithmetic expression which contains two or more operators, the order of their application during evaluation is important. In BASIC the order of evaluation is implicitly specified by a hierarchy of precedence among the operators. This hierarchy, adopted from the conventional rules of mathematics, is illustrated by the following table:

**	↑	(highest level)
unary -		unary +
*	/	
+	-	
MIN	MAX	(lowest level)

When contemplating two operators which appear at different levels in this table, the operator at the higher level is chosen for execution first. In the arithmetic expression $A+B*C$ the multiplication precedes the addition. When two operators appear at the same level, the operators are executed from left to right. In $A+B-C$ the addition precedes the subtraction. In some cases the implicit order of evaluation is not the one desired. Note that one form of a *primary* is a parenthesized *numeric expression*. Any portion of a *numeric expression* which by itself is also a legal form of *numeric expression* can be enclosed in parentheses. The enclosed portion becomes a *primary*, forcing its evaluation to precede execution of any operators in the surrounding *numeric expression* which might otherwise take precedence. Parentheses can be nested to any depth required to override the implicit order of evaluation.

Example:

$A+B/C \text{ MIN } -D*+E**F$ is evaluated in the same order as
 $(A+(B/C)) \text{ MIN } ((-D)*(+(E**F)))$

Arithmetic expressions can be combined into a more complex form of *numeric expression* called a 'relational expression'. A relational expression is a sequence of one or more arithmetic expressions (usually two), where each arithmetic expression (except the last or only one) is syntactically separated from its successor by a *relational operator*. During evaluation a *relational operator* is interpreted as a binary operator which produces a value of 'true' or 'false'. The most common use for a relational expression is within an IF statement.

Example:

```
IF A+B <= C+D THEN 300
```

The most general form of *numeric expression* is the 'logical expression'. Three logical operators are available: AND, OR, and NOT. The first two are binary operators used to combine relational expressions into more complex decisions. The result of executing OR is 'true' unless both of its operands have the value 'false'. The result of executing AND is 'false' unless both of its operands have the value 'true'. NOT is a unary operator and has the same precedence as unary '-' and unary '+'. NOT performs a logical negation; its result is 'false' if the value of its operand is 'true' and its result is 'true' if the value of its operand is 'false'.

Example:

```
IF (A+B>C AND X=Y) OR NOT D<= E+2 THEN 45
```

Although logical operators are described in terms of manipulating 'true' and 'false', the values are actually represented within BASIC as numbers. If the result of a logical operator or *relational operator* is 'true', it produces the number 1. If the result is 'false' it produces the number 0. The operand of a logical operator will be interpreted as 'true' if its value is non-zero; it will be interpreted as 'false' if its value is 0. Logical operators can use any *numeric expression* as their operand and both logical operators and *relational operators* can appear within *numeric expressions* enclosed in parentheses and used as primaries of arithmetic expressions. The complete operator hierarchy for *numeric expressions* is the following:

```
**      ↑      (highest)
unary +   unary -   NOT
*      /
+      -
MIN      MAX
<= < = > >= <> #
AND
OR      (lowest)
```

The same rules for order of evaluation given under the discussion of arithmetic expressions apply to the expanded table. Parentheses can be used to override the implicit order of evaluation at any level of a *numeric expression*. Note that $A < B < C$ is not equivalent to $A < B$ AND $B < C$. The format evaluates $A < B$ to a 1 or 0 and then compares that result to C. The latter evaluates $A < B$ to 'true' or 'false' (1 or 0) and $B < C$ to 'true' or 'false' (1 or 0) and then produces the logical AND of the two results.

Example:

```
A AND B<C+D OR NOT E    is evaluated in the same order as
(A AND (B<(C+D))) OR (NOT E)
```

NUMERIC SIMPLE VARIABLE

A *numeric simple variable* is a single alphabetic character (A through Z) optionally followed by a single digit (0 through 9). It is used to reference a number, which is also referred to as its value. The value can be accessed or altered by several BASIC language constructs. Normally a *numeric simple variable* is uniquely identified by its name and its value is available to any statement within a program. However, a special case exists when a name of this form appears as the parameter of a DEF statement. In this context the value of the name is assigned by a reference to the associated function name and is accessed by use of the parameter name as a *primary* within the numeric expression of the DEF statement. The same name can appear as the parameter of different DEF statements within the same program and also as a *numeric simple variable* in other statements. A parameter is recognized as unique within the context of its own DEF statement; thus all appearances of its name outside of that statement are independent. The values associated with these other appearances are maintained separately and do not interact with the value of the parameter.

Example:

A9, P3, G, C, F0, Z2

NUMERIC VARIABLE

A *numeric variable* is either a *numeric simple variable* or a *subscripted variable*.

Examples:

H2, F(3), M(I,J+2), Z

PHYSICAL LENGTH

The *physical length* is the maximum number of characters that the *string value* of a *string simple variable* can contain.

Example:

100 DIM A\$(30), P1\$(72) (defines the *physical lengths* of A\$ and P1\$ as 30 and 72, respectively)

PHYSICAL SIZE

The *physical size* is the maximum size an *array* can attain. This is the same as the size established either explicitly in a DIM statement or COM statement or implicitly in the absence of an explicit specification.

Examples:

10 DIM A(45), B(3,2) (defines the physical size of A, B, and M as 45, 6, and 30, respectively)
20 COM M(30)

PRIMARY

A *primary* is a *numeric constant*, a *numeric variable*, a *function reference*, or a *numeric expression* enclosed within parentheses. *Primaries* are used to supply the numbers which are combined and manipulated by the operators of a *numeric expression*.

PROGRAM DESIGNATOR

A *program designator* identifies a *program name*. It is either a string constant or a string variable whose value is a program name.

Examples:

"PROGA"

A\$ where value of A\$ is *program name*

PROGRAM NAME

A *program name* is a symbol composed of 1 to 6 letters and/or digits. It is used to identify a specific program in a library. In some contexts the name can be optionally qualified (\$, *, or .idcode). A preceding dollar-sign (\$) indicates that the named program is to be found in the system library. A preceding asterisk (*) indicates that the named program is to be found in the group library. Following the name with a period (.) and idcode indicates that the named program is to be found in the library of that idcode.

Examples:

Valid Program Names		Invalid Program Names
PROG	YRPRG.F932	PROGRAM (more than 6 characters)
B	\$DRAG	PROG/2 (/ is not a letter or digit)
TDE 200	*GPROG	

RECORD NUMBER

A *record number* is a *numeric expression* which evaluates to a number specifying a particular record within a BASIC formatted file. The *numeric expression* is evaluated and rounded (if necessary) to an integer. The result must be greater than or equal to 1 and less than or equal to the number of records in the file.

RELATIONAL OPERATOR

A *relational operator* is one of the symbols:

- < (less than)
- <= (less than or equal to)
- = (equal)
- >= (greater than or equal to)
- > (greater than)
- <> (unequal)
- # (alternate for <>)

Each symbol, when executed, compares the value of the operand to its left with the value of the operand to its right and returns the value 'true' (1) if they satisfy the relation

represented or 'false' (0) if they do not. *Relational operators* can be used either to compare the values of two strings (within an IF statement only) or to compare two numbers within a *numeric expression*. Note that the symbol used to test for equality (=) is the same as the symbol used to specify assignment in a LET statement. If this symbol appears more than once within a LET statement, there can be confusion as to which meaning it has (multiple assignment or test for equality). In such cases the rule is that, starting from the beginning of the statement, all appearances of '=' are assignment operators until one of them is followed by a construct which is not a *numeric variable*. Any remaining appearances are the *relational operator*.

Examples:

A = B > C	A is set to 1 or 0 depending on truth of relational expression B > C
A = B = C > D	A and B are both set to 1 or 0 depending on C > D
A = (B = C) > D	A is set to 1 or 0 depending on evaluation of relation (B = C) > D

RETURN VARIABLE

A *return variable* is a *numeric variable* used in the context of a statement which returns status information to a BASIC program. The statement indicates the results of its execution by assigning a value to the *return variable*. The program can then test the result of such statements by accessing the value.

Example:

100 ASSIGN "FIL1", 3, Z (Z is a *return variable*)

SOURCE STRING

A *source string* is a *literal string* or a *string variable* used in a context where it supplies a *string value*. The BASIC language supplies a number of constructs which manipulate a *source string*. For example, in the following statements, the *string variable* A\$ and the *literal string* "ABC" are *source strings*.

```
10 LET X$=A$
20 PRINT "ABC"
30 PRINT UPSS(A$)
```

If the *source string* is a *literal string* its value and length are simply those of the string represented textually. For example,

21 PRINT "ABCDE" the *source string* value is ABCDE, its length is 5

If the *source string* is a *string simple variable* then the value and length of the source string are the current value and current *logical length* of the string referenced by the variable name. This form returns the entire current string as the accessed string. For example,

```
22 LET A$="ABC"
23 PRINT A$      the entire current value of A$ is accessed
```

If the *source string* is a *string variable* with a single *substring designator* then that part of the current value preceding the specified character position is ignored. The value of the accessed string consists of any remaining characters in the current value and its length is the number of characters remaining. If the specified character position exceeds the current *logical length* by one position then the accessed string is a null string (a string of zero length containing no value). If the specified position exceeds the logical length by more than one position, an error occurs.

Examples

Assume A\$ has the value "ABC" and a *logical length* of 3:

24 PRINT A\$(2)	The value BC is printed; it has a <i>logical length</i> of 2
25 PRINT A\$(4)	A null string is accessed; nothing is printed.
26 PRINT A\$(5)	An error occurs; position 4 is undefined.
27 PRINT A\$	The value ABC is printed; A\$ has not been altered.



If the *source string* is a *string variable* with a double *substring designator* then the *string* starting with the first character position and continuing through the second is accessed. If the last character position is greater than the *logical length* of the current value, the non-existent characters are interpreted as blanks. If the first character position is greater than the current *logical length*, the accessed string consists entirely of blanks. The length of the accessed string is determined by the substring designator and is equal to the second value – first value + 1. If this length is equal to zero, the accessed string is a null string and none of the current value is used. If less than zero, the access string is undefined causing an error.

Examples:

Assume A\$ has the value "ABCDE" with a logical length of 5:

31 PRINT A\$(1,3)	The value ABC is printed; it has a logical length of 3.
32 PRINT A\$(4,8)	The value DE^^^ is printed with a logical length of 5.
33 PRINT A\$(6,15)	A string of 10 blanks is printed.
34 PRINT A\$(4,3)	Accessed string is null; nothing is printed.
35 PRINT A\$(4,2)	Accessed string undefined; error condition

STATEMENT NUMBER

A *statement number* is an integer in the range 1 to 9999 inclusive. Each statement in a *program* is preceded by a unique statement number. It is used both to indicate the proper order of the statement relative to other statements in the program and as a label by which other statements can reference its statement during execution.

Example:

100 A=B (100 is the statement number)

STRING

A *string* is an ordered collection of characters which taken all together constitute the *string value*. The maximum length of a *string*, its *physical length*, can be specified in a DIM or COM statement; if not specified, its *physical length* is one character. The number of characters in a *string* can be between 0 and 255; the minimum that can be specified in a DIM or COM statement is one character. A zero length string is known as a null string and it has no value. The actual number of characters used in a *string* may be less than the number dimensioned; the length used is known as the *logical length*.

The *string value* is expressed as a *string expression*. A *string expression* can be a *literal string*, a string-valued *function*, or a *string variable*. A *string variable* can be either a *string simple variable* or a *string simple variable* followed by a *substring designator*. When a *string* is used as a *destination string*, it can be a *string variable*; used as a *source string* it can be a *string variable* or a *literal string*.

Examples:

```
10 DIM A$(20)
20 A$="THIS IS A STRING"
30 PRINT A$(11,17);"A$"
RUN
STRING A$
```

STRING EXPRESSION

A *string expression* is one of several BASIC constructs used to supply a *string value*. A *string expression* may be a *string variable*, a *literal string*, or one of the two string-valued *functions* CHR\$ and UPS\$. Any *string expression* can be used as a *source string*, only a *string variable* can be used as a *destination string*. The specific interpretation and use depends on the context in which it appears.

Examples:

```
"ABC"
"AB" '67
A$
A$(3,6)
CHR$(I)
UPS$(B$)
```

STRING LENGTH

(Refer to *string*)

STRING SIMPLE VARIABLE

A *string simple variable* is a single alphabetic character (A through Z), optionally followed by either 0 or 1, followed by a dollar-sign (\$). It is used to reference a string. The value and length of the referenced string can be accessed or altered by several BASIC language constructs.

Examples:

A\$, P1\$, F\$, Z0\$

STRING VALUE

(Refer to *string*)

STRING VARIABLE

A *string variable* is either a *string simple variable* or a *string simple variable* qualified by a *substring designator*.

Examples:

G0\$, D\$ *string simple variables*
 A\$(4),Q1\$(1,LEN(Q\$)),M\$(I,J) *string simple variables qualified by substring designators*

SUBSCRIPTED VARIABLE

A *subscripted variable* is an *array name* followed by either one or two subscripts enclosed in parentheses. It is used to reference the value of an *array element*. A subscript is a numeric expression which is evaluated and rounded (if necessary) to an integer. An element of a one-dimensional *array* is referenced by following the *array name* with a single subscript within the parentheses. It must evaluate to an integer from 1 to the *logical size* of the *array*. An element of a two-dimensional *array* is referenced by following the *array name* with two subscripts, separated by a comma, within the parentheses. The first must evaluate to an integer from 1 to the current row size; the second must evaluate to an integer from 1 to the current column size.

Examples:

F(I,J+3), S(M1*F/2), M(32)

SUBSTRING DESIGNATOR

A *substring designator* is a modifier which can follow a *string simple variable* to designate either an initial character position or a range of character positions. The general form is:

(*numeric expression* [,*numeric expression*])

If only one *numeric expression* is used then the designator is single, specifying an initial character position in a range terminated by the last character in the string variable. If both *numeric expressions* are used then the designator is double, that is, the first *numeric expression* specifies the first character position in the range, the second expression specifies the last character position in the range.

Examples:

A\$(N+1), A1\$(16),B\$(LEN(A\$))	single <i>substring designators</i>
Z\$(ABS(X),ABS(Y)), Z1\$(30,30), M\$(1,20)	double <i>substring designators</i>

Each *numeric expression* is evaluated and, if necessary, rounded to an integer before use. The value of the first or only expression must be an integer in the range 1 through 32767. In practice any value above 255, the maximum physical length of a *string*, has little utility. The second expression may be zero only in the special case of a null substring where the first expression is 1, i.e., (1,0).

The number of character positions in the range specified by a double *substring designator* is the second value – first value +1. Thus the second value is normally greater than or equal to the first value. As a special case, the second value can be 1 less than the first value specifying a range of zero characters. Such a zero-length string is a null string that has no value. An error results if a negative range is specified (second value is 2 or more less than the first value) or if the range is greater than 255 characters, the maximum *physical length* of a *string*. Other than these restrictions, the legality of a particular *substring designator* depends on the context where it is used.

Examples:

Assume A\$ has the value "ABCDE":

A\$(2)	positions 2 through 5 produce a range of 4 characters
A\$(2,7)	positions 2 through 7 produce a range of 6 characters
A\$(2,1)	A null string is referenced
A\$(3,1)	An error results

STATEMENTS AND FUNCTIONS

All the statements and functions in the 2000 BASIC language are described in alphabetic order in the remainder of this section. Each description consists of the general form of the statement or function, a description of its operation, and one or more examples. The descriptions use the terms defined in the previous pages. Whenever such terms appear they are printed in *italics*. Terms that are defined within the statement or function description itself are printed in ***bold italics***. Optional parameters are enclosed in square brackets.

Refer to table 11-1 for a list of the statements and table 11-2 for a list of the functions in the HP 2000 BASIC language.

Table 11-1. Statements

ADVANCE	IMAGE	MAT READ#
ASSIGN	INPUT	MAT Scalar multiplication
CHAIN	LET	MAT...TRN
COM	LINPUT	MAT.....ZER
CONVERT	LINPUT#	NEXT
CREATE	LOCK	PRINT
DATA	MAT addition & subtraction	PRINT USING
DEF	MAT assignment	PRINT#
DIM	MAT.....CON	PRINT# USING
END	MAT.....IDN	PURGE
ENTER	MAT INPUT	READ
FILES	MAT.....INV	READ#
FOR and NEXT	MAT multiplication	REM
GOSUB and RETURN	MAT PRINT	RESTORE
GO TO	MAT PRINT#	STOP
IF.....THEN	MAT PRINT USING	SYSTEM
IF END	MAT PRINT# USING	UNLOCK
IF ERROR	MAT READ	UPDATE

Table 11-2. Functions

ABS	INV	SIN
ATN	ITM	SPA
BRK	LEN	SQR
CHR\$	LIN	SYS
CON	LOG	TAB
COS	NUM	TAN
CTL	POS	TIM
EXP	REC	TRN
IDN	RND	TYP
INT	SGN	UPS\$
		ZER

ABS Function

General Form:

ABS (*numeric expression*)

The ABS function is a numeric valued function which returns the absolute value of the *numeric expression*.

The absolute value of a number is that number itself if it is positive and the negative of that number if it is negative. For example, $ABS(33) = 33$, and $ABS(-33) = 33$.

Example:

```
20 PRINT ABS(N/D*S)
```

ADVANCE Statement

General Form:

ADVANCE # *file number*; *skip count*, *return variable*

The ADVANCE statement causes the pointer for the specified file to be advanced past the number of items specified by the *skip count*.

The *skip count* is a *numeric expression* which, when evaluated and rounded to an integer, specifies the number of items to be skipped. The number of items to be skipped is limited to 32767 and may not be negative. If the statement is executed successfully, the *return variable* is set to 0. If the statement encounters an EOF (end-of-file mark) before the specified number of data items have been skipped, the *return variable* will be set to the number of items yet to be skipped. If the statement encounters an EOR (end-of-record mark) before the specified number of data items have been skipped, skipping continues into the next record. The ADVANCE statement cannot be used on an ASCII file; if attempted, the program will be terminated. Specification of an invalid *file number* will result in an error. (Refer to the FILES statement for an explanation of file number.)

Example:

```
10 ADVANCE #3;5*X+1,Z
```

ASSIGN Statement

General Form:

ASSIGN *file designator*,*file number*,*return variable* [,*mask*] [,*restriction*]

or

ASSIGN *,*file number* [,*return variable*]

The ASSIGN statement is used to assign a *file designator* to a *file number* reserved by the FILES statement and to open the specified file. For example, ASSIGN "AFILE," 3, VAR assigns the file name AFILE to the position reserved for the file number 3 in the FILES statement.

If a file is already associated with the *file number* used in an ASSIGN statement, then that file is closed. The result of the assign operation is returned in the *return variable*. The *file designator* is a *source string* whose value is a *file name*. The *file name* referenced may be a system library file (*\$ name*), a group library file (**name*), or a file resident (saved) in any other user's library (*name.idcode*). To reference a file with *name.idcode*, that idcode must have the PFA (Program/File Access) capability. The *file number* is a *numeric expression* which, when evaluated and rounded to an integer specified the *file number* (1–16).

If an asterisk (*) is used in place of the *file designator*, the file previously associated with the file number is closed. If the file has already been closed, the statement is ignored.

Refer to Section V for a description of file usage.

Example:

```
100 ASSIGN *, I, Z
```

After ASSIGN is executed, a value is returned in the *return variable*. The return values and their meanings are given below.

Return Value	Meaning
0	file is available for read and write
1	file is available for read only
2	file is available for read only (it belongs to another user and is protected)
3	file does not exist or is not accessible
4	file number is out of range (it does not correspond to one of the positions reserved by a FILES statement)
5	no buffer space is available for the file
6	file is not available for read or write because of another user's current access
7	specified restrictions not possible
8	file is available for write only

Basic Language Reference

ASSIGN Statement (Cont)

If the value returned is 3, 4, 5, 6, or 7 the file is not opened, any access to the file number causes the program to be terminated with an error. If the returned value is 1 any attempt to print onto the file causes a terminal error. If the returned value is 8, any attempt to read the file causes a terminal error. Other references to the file assigned that file number are legal.

The optional *mask* is a *source string* used to encode or decode BASIC formatted file data (refer to Section V). When using a *mask*, all data read or written to the file will be modified by the *mask*, making the data unintelligible to programs which use the file without the same *mask*. In order to extract data from a file, the same *mask* must be used to read the data as was used to print the data to the file. Numeric zeros, data types, EOR (end-of-record) marks, and EOF (end-of-file) marks are not affected. A *mask* used with an ASCII file is ignored.

Example:

```
100 ASSIGN "FIL1", I, Z, M$
```

The optional *restriction* is a two-letter code used to specify any dynamic access restrictions on the file. Dynamic access refers to access by more than one user whether more users are currently accessing the file or may subsequently access the file. Restriction codes are as follows:

Code	Dynamic Access
RR	Read and write restriction; no subsequent user can access the file while the file is open.
WR	Write restriction; subsequent users can read from, but not write to the file while the file is open.
NR	No restriction; subsequent users can read and write file data while the file is open. (As long as the file is multiple write access — MWA.)

If the *restriction* parameter is omitted for a SWA (single write access) file, the file is opened with the WR (write restriction) restriction. If this fails because another user has write access, NR (no restriction) is used. For MWA (multiple write access) files, the NR code is used when *restriction* is omitted.

The specified restriction remains in effect as long as the file is open. If another program has opened the file with sufficient restrictions, the ASSIGN statement returns a value of 6 or 7 in the *return variable* and the file is not opened.

For a file located in the user's log-on account that is not currently in use, read/write access is granted. ASCII Files are opened with read-only or write-only access depending on the device. If the file is in use, and it is single write access (SWA), read-only access is granted unless the other user has applied the RR restriction when assigning the file, in which case no access is granted. If the file is in use and it is multiple write access (MWA), read/write access is granted unless the other user has applied the WR restriction when assigning the file, in which case read-only access is granted. If the other user has applied the RR restriction, no access is granted.

ASSIGN Statement (Cont)

For a BASIC formatted or ASCII disc file located in a library other than (outside) the user's log-on account, that is *not* currently in use, static access is granted as follows. Note that static access refers to access by a single user.

File Status	Static Access
Unrestricted	Read/write
Protected	Read only
Locked	No access unless the program accessing the file is saved in the same account as the file and the program is locked, in which case read/write access is granted.
Private	No access

You may not access an ASCII file located in a library other than your log-on account with the following two exceptions: 1) ASCII disc files, or 2) Locked ASCII files in the A000 library accessed by a locked program also saved in the A000 library.

For any file currently in use, located in a library other than the account you are using, the access granted is the same as described above, modified by any dynamic restrictions (NR, WR, RR) that the other user has applied either explicitly or by default.

A locked or private program in a group library account with the FCP (File Create/Purge) capability which has been executed or chained-to, has static read/write access to locked BASIC formatted files having the PFA (Program/File Access) capability, located in any of the accounts of that group.

In the example below, files are assigned for each file number associated with an * in the FILES statements. A write restriction (WR) on XFILE prevents other users from writing on that file. A read and write restriction (RR) on DFILE prevents other users from having any access to the file. There are no access restrictions (NR) on FF1. The mask "ABZ1" is used to encode the data in file X. File X is closed in line 90. In line 100, AFILE is closed and file CC is opened as file number 1. The zeros in the numeric variables indicate that each file was available for reading and writing when it was opened.

Examples:

```

10 FILES AFILE,BFILE,*
20 FILES AA,BB,*,*
30 ASSIGN "XFILE",3,X1,WR
40 ASSIGN "DFILE",6,D1,RR
50 ASSIGN "FF1",7,N,NR
60 LET X$="X"
70 ASSIGN X$,8,X,"ABZ1"
80 PRINT X1,D1,N,X
90 ASSIGN *, 8
100 ASSIGN "CC",1,C1
110 PRINT C1
120 END
RUN
0      0      0      0
0

```

ATN Function

General Form:

ATN (*numeric expression*)

ATN is a numeric-valued function which returns the arctangent of the *numeric expression*.

The value of the function is the angle (in radians) whose tangent is the *numeric expression*.

Example:

100 A = ATN(B)

BRK Function

General Form:

BRK (*numeric expression*)

The BRK function enables or disables the BREAK key capability of a terminal.

The value of *numeric expression*, when evaluated and rounded to an integer, will have the following effect:

- Value less than zero returns the status of the BREAK capability.
- Value equal to zero disables the BREAK capability.
- Value greater than zero enables the BREAK capability.

BREAK CAPABILITY. For a program running at a terminal, the BREAK key capability of the terminal can be disabled or enabled by execution of the BRK function within the program. At the beginning of program execution, the break capability is enabled by default except for the special case of the HELLO program where the BREAK key is initially disabled. If disabled, it remains disabled until program execution is completed, the program terminates because of an execution error, the user hangs up causing him to be logged off the system, the BREAK command is entered by the system operator, or until the BRK function is executed with an argument greater than zero.

BRK Function (Cont)

Because program execution may be completed before all output is complete, care should be taken when re-enabling the break capability to ensure that program output will not be interrupted and lost. Either an INPUT statement or an ENTER statement can be included in the program just prior to the statement containing the BRK enable function. This will cause the program to pause until output is complete before continuing execution. For example, the following program segment disables the break capability and prints the value of "I" twenty times. On encountering the ENTER statement, the program pauses until printed output is complete before execution continues from statement 30. The same effect can be achieved with the PAUSE command (Refer to Section X).

Examples:

```
5 Y=BRK(0)
10 FOR I=1 TO 20
15 PRINT "I=";I
20 NEXT I
25 ENTER 1,A,B
30 Z=BRK(1)
99 END
```

VALUES RETURNED BY BRK FUNCTION. For arguments equal to or greater than zero, the value returned depends on the previous condition of the BREAK capability. This value will be 1 if the capability was previously enabled, or 0 if the capability was previously disabled.

To find the current status of the BREAK capability, enter an argument less than zero. If currently enabled, a 1 is returned. If currently disabled, a 0 is returned.

If a program is in an infinite loop during execution and the BREAK capability is disabled, the system operator can enter a BREAK command to enable the BREAK capability. Once the system operator enables the BREAK capability, the running program may not disable BREAK until program termination.

Examples:

```
935 LET B = BRK(0)
940 Z = BRK(A+M)
945 PRINT BRK(Y)
```

CHAIN Statement

General Form:

CHAIN [*return variable*,]*program designator*[*numeric expression*]

The CHAIN statement causes the current program to terminate and the program referenced by *program designator* to be executed.

The *program designator* is a source string (string constant or variable) whose value is a *program name*. The *program name* may be a system library program (\$name), or a program in any other user's library (name.idcode). To reference a program with name.idcode, that idcode (account) must have the PFA (Program/File Access) capability. You may always CHAIN to a program located in the account you are using, regardless of its status (i.e., locked, private, protected, or unrestricted).

For a program located in a library other than the account you are using, the access granted is as follows:

Program Status	Static Access
Unrestricted	May chain, optionally specifying a statement number.
Protected	May chain, optionally specifying a statement number.
Locked	May chain, but no statement number may be specified unless the current program is saved in the same library as the chained-to-program. May chain to statement number if FCP requirements are met (refer to Section VIII for FCP discussion).
Private	Chain not allowed.

The optional *numeric expression* following the *program designator* can be used to define the *statement number* in the destination program where execution will begin. If not specified, the new program will begin execution at the first program statement. The *numeric expression* is evaluated and rounded to the nearest integer to obtain the starting *statement number*.

If the optional *return variable* is provided and chain operation is successful, the *return variable* is set to 0. This value is accessible in the destination program if the *return variable* is specified as common, although its utility is limited.

If a chain operation is unsuccessful, the *return variable* is set to a value of 1, 2, or 3, depending on the type of error. A list of *return variable* meanings is as follows:

Return Value	Meaning
0	Successful
1	Bad statement number specified (less than 1 or greater than 9999)
2	No access permitted to named program
3	Chain not permitted

CHAIN Statement (Cont)

When a *return variable* is used and a chain operation is not successful, execution of the current program continues with the statement following the CHAIN statement. Unsuccessful chain operations without a *return variable* cause the current program to be terminated with an error.

If a chain operation is successful, all files in the chaining program are closed, even if they appear in FILES or ASSIGN statements in the destination program. No other user, however, may gain access to files in the chaining program until any FILES statements in the chained-to program are executed. Also, any variables listed in a COM statement are transferred (see the COM statement elsewhere in this section).

Before execution of the destination program can begin, it must be compiled. Programs may be stored in the semi-compiled form to speed execution (see the CSAVE command in Section X).

Examples:

```
20 CHAIN "PROG2"  
50 CHAIN V$  
97 CHAIN "ABC",A  
150 CHAIN "MELVIN",80  
200 CHAIN N$,Q+14  
230 CHAIN A$,110  
240 CHAIN R,"LIBROC.A001",10
```

CHR\$ Function

General Form:

CHR\$ (*numeric expression*)

The CHR\$ function returns a single ASCII character that has the value of *numeric expression*.

Each character in a string is internally represented by one byte consisting of eight bits. These eight bits allow 256 different characters to be represented. The representation of the first 128 of these characters (0 — 127) is in accordance with the ASCII standard. The last 128 characters have no predetermined meaning (have only a positive value from 128 to 255). All 256 characters can be referred to by a decimal number in the range of 0 to 255. Appendix A presents the decimal equivalents for each of the representable characters (i.e., A = 65, B = 66, etc).

The CHR\$ function returns a single character (byte) which corresponds to the position in the ASCII code table (Appendix A) for the argument of the function (the *numeric expression*). The *numeric expression* is evaluated and rounded to an integer which should be in the range of from 0 to 255. If the integer is not within this range, then the program terminates with an error. For example, CHR\$(65) and CHR\$(30 + 35.4) both yield the character "A".

The CHR\$ function may be used only on the right side of a string assignment statement; string IF statement; or as a print list item.

Examples:

```
200 PRINT CHR$(2*(1+B))
300 A$(1,1) = CHR$(J)
```


COM Statement

General Form:

COM common list

The COM statement lists variables to be passed between programs linked by the CHAIN statement.

A *common list* is one or more common elements separated by commas. A common element is a *numeric simple variable*, *string simple variable*, *string simple variable (length)*, *array name (number of rows)*, or *array name (number of rows, number of columns)*.

Example:

```
100 COM A,B(10,10),B$(200)
```

Several programs may be run sequentially, all accessing and possibly changing data in the common area. Program control must be transferred with the CHAIN statement. Merely getting and running the next program will not preserve the common area. COM statements must be the lowest numbered statements in the program. The transfer of data values between variables in different programs is done according to the order of the variables in the COM statement. For example, if one program has as its first statement

```
10 COM A,B1,C$(10)
```

and a second program has as its first statements

```
12 COM X
14 COM Y,Z$(10)
```

and the first program chains to the second, the current values of variables A,B1, and C\$ will be used to initialize the variables X, Y, and Z\$ respectively. Note that you are free to use the variable names A,B1 and C\$ for a different purpose in the second program.

Array and string variables used in COM statements must be dimensioned in the COM statement and may not appear in DIM statements. The corresponding variables in the chained-to program must also be dimensioned in the COM statement and must be dimensioned to the same values as those in the first program. In order to access a common variable, all preceding common variables must correspond in type and dimension to those in the first program. You need not have the same number of common variables in both programs.

The values of common variables are lost when a program terminates (whether due to an END or STOP statement; to an execution error; or to the use of the BREAK key).

Examples:

```
100 COM A,Q0$ (255)
200 COM C,R1,C3
300 COM F(30,12), B$(10), Z(14)
```

CON Function

(Refer to MAT . . . CON)

CONVERT Statement

General Form:

CONVERT *numeric expression* TO *destination string*

or

CONVERT *source string* TO *numeric variable* [, *statement number*]

The CONVERT statement is used to change a *numeric expression* to a string of characters that represent the value of the expression. It may also be used to convert a *source string* to an equivalent numeric value.

NUMERIC TO STRING. When converting numeric values to string values, the conversion is the same as that used by the system to list numeric data (using the LIST command). The output string will not contain embedded blanks unless a blank replaces a plus sign in an unsigned positive number. The string resulting from conversion follows the rules for a *destination string*.

Example:

```
100 DIM A$(3)
150 A=10
170 B=15
200 CONVERT A+B TO A$
220 PRINT A$
```

This will result in the string " 25" being printed as three characters as in the normal numeric format of a listing.

STRING TO NUMERIC. When converting strings to numeric values, the string must represent a valid numeric constant. If not, the program will be terminated unless an optional statement number is specified. In this case, an invalid numeric value will transfer control to that statement. Blanks are permitted in the string.

Examples:

```
200 CONVERT P1$(21,30) TO F(7)
300 CONVERT A$ TO X,500
400 CONVERT B$ TO M1
```

COS Function

General Form:

COS (*numeric expression*)

The COS function is a numeric valued function which returns the cosine of the *numeric expression*. The *numeric expression* is interpreted as being in radians. If the absolute value of the *numeric expression* exceeds approximately 102900, your program will be terminated with an error.

Example:

```
500 LET B = COS(3.1415/X)
```

CREATE Statement

General Form:

CREATE *return variable*,*file designator*,*file length* [, *record size*]

The CREATE statement is used to create a BASIC formatted file from an executing program.

The CREATE statement can be used only to create a BASIC formatted file and cannot be used to create an ASCII file. The *return variable* is set to a value which indicates the result of the execution of the statement. The values which may be returned and their meanings are as follows:

Return Value	Meaning
0	The file was created successfully
1	A file already exists with the same name
2	Invalid file name, no such account, invalid access, invalid file length or record size
3	No space in the account
4	No space in the system

The *file designator* is a *source string* whose value is a *file name*. The *file length* is a *numeric expression*, which, when evaluated and rounded to an integer, specifies the number of records to be created in the file. File size may vary from a minimum of one record to a maximum of 32,767 records. (This value may be limited by account and system restrictions.) The optional *record size* must be a *numeric expression* which, when evaluated and rounded to an integer, gives the file record size in words. The *record size*, if specified, must be between 64 and 256 words. If not specified, the record size is set to 256 words. Regardless of record size, a block is always 256 words. The status of any file created with the CREATE statement is locked.

A locked or private program saved in a group library account having the File Create/Purge (FCP) capability which has been executed or chained-to may create a BASIC formatted file in any of the account libraries which are members of that group having the Program/File Accessibility (PFA) capability.

Note that the CREATE statement does not open a file for access. The file must still be opened using a FILES or ASSIGN statement.

Examples:

```
10 CREATE N,"MYFILE",200
20 CREATE M,"HERFIL",500,64
30 CREATE P,A$,100
40 CREATE Q,B$,X**2,J
```

CTL Function

General Form:

CTL (numeric expression)

The CTL function can be included in print operations to provide hardware control and mode switching for ASCII file devices. This function may be included in any of the print or file print statements (PRINT, MAT PRINT, PRINT USING, or PRINT# MAT PRINT#, PRINT# USING). CTL is ignored for BASIC formatted files and your terminal. When used for an input device, items to be printed must not be included in the print list since printing data to an input device causes a fatal error.

CTL functions can be included in operations directed to any device. For instance, the *OUT=file name* form can be used to direct print operations to an output device. When the particular function is not defined for the device, CTL is ignored. For example, all CTL functions directed to a terminal are ignored.

A comma following a CTL function in a print operation is treated like a semicolon (refer to *print delimiter* under the PRINT statement description). Any characters specified in print items prior to a CTL function are printed before CTL is executed.

The argument of the control function is a numeric expression that when evaluated and rounded results in an integer. This value is used to select a device command to be sent to the ASCII device. The functions defined by the *numeric expression* are listed in the following table.

Examples:

1. FILE-ASCII,LP1

```
10 FILES ASCII
20 DIM A0$(50)
25 A0$= "THIS IS THE FIRST ITEM IN FILE ASCII"
30 PRINT#1;A0$,CTL(14),"END OF DATA"           skip one line between strings
```

2. FILE-PRINTR,LP2

```
RUN *OUT=PRINTR*
100 DATA 100,200,"A","B",1,2
110 READ N,M,A$,B$,W,X
120 PRINT N,M,CTL(3),A$,B$,CTL(3),W,X           skip to channel 3 after printing M and B$
```

3. FILE-READR,PRO

```
100 FILES READR
110 PRINT#1;CTL(33)                             prepare to read binary data
120 LINPUT#1; A$
```

CTL Function (Cont)

Defined Uses of the CTL Function

VALUE OF CTL ARGUMENT	EFFECT
Line printers use arguments 1 to 13	
1	Skip to channel 1 (normally top of form)
2	Skip to channel 2 (normally bottom of form)
3	Skip to channel 3 (normally next line)
4	Skip to channel 4 (normally next double line) ¹
5	Skip to channel 5 (normally next triple line) ²
6	Skip to channel 6 (normally next half page)
7	Skip to channel 7 (normally next quarter page)
8	Skip to channel 8 (normally next sixth page)
9	Skip to channel 9 (installation defined) ³
10	Skip to channel 10 (installation defined) ³
11	Skip to channel 11 (installation defined) ³
12	Skip to channel 12 (installation defined) ³
13	Suppress spacing (suppresses paper advance) ⁴
14	Skip one line (complete current line)
15	Skip two lines
16	Skip three lines
Magnetic tapes use arguments 20 to 24	
20	Skip forward to tape mark (the tape is positioned just before the next tape mark)
21	Skip to next file (the tape is positioned just past the next tape mark; the end of file condition occurs if the end-of-tape mark is read before the next end-of-file)
22	Skip backward to tape mark (the tape is positioned just past the preceding tape mark)
23	Skip to preceding file (the tape is positioned just past the second preceding tape mark; the end of file condition occurs if the beginning-of-tape mark is read before the first preceding end-of-file mark)
24	Rewind to beginning of tape mark
ASCII disc files use argument 24	
24	Logically rewind file (reposition the file pointer to the beginning of the file) ⁵
Paper tape punches use arguments 30 to 33	
30	Punch data with even parity and with X-OFF, RETURN, and LINE FEED as record separator (this is the default mode) ⁶
31	Punch data with odd parity and with X-OFF, RETURN, and LINE FEED as record separator ⁶
32	Punch data with no parity and with the X-OFF, RETURN, and LINE FEED as record separator ⁶
33	Punch data with no parity and no record separators (this can be used for binary output) ⁶

CTL Function (Cont)

Computer
Museum

VALUE OF CTL ARGUMENT	EFFECT
Paper tape readers use arguments 30 to 33	
30	Read data with even parity and with RETURN as record separator
31	Read data with odd parity and with RETURN as record separator
32	Read data with parity ignored and with RETURN as record separator (default mode)
33	Read data with parity ignored and no record separators (this can be used for binary input)
Reader/Punch/Interpreter uses arguments 40 through 54; card readers (2892 and 7261) use arguments 40 through 42	
40	Read ASCII card image (default)
41	Read column binary image
42	Read EBCDIC card image
43	Feed a card into reader
44	Punch hollerith (ASCII) code on card (default)
45	Punch column binary code
46	Select hopper 1 (default)
47	Select Hopper 2
48	Select stacker 1 (default)
49	Select stacker 2
50	Punch and print same data (default)
51	Punch only
52	Print only
53	Punch and print different data
54	Select stacker overflow mode
The ASCII RJE devices designated as JL (job lister), JP (job punch) or JT (job transmitter) use codes 60 and 61	
60	Translate ASCII to EBCDIC or EBCDIC to ASCII (default)
61	Do not translate data
The link terminal uses arguments 70 through 74	
70	Send HPIB message
71	Change state of indicator lights
72	Change terminating mode of special function key
73	Enable display unit
74	Disable display unit
Arguments 75 to 81 are reserved for future use by the link terminal.	
Notes to CTL argument functions:	
¹ Skipping to a double space line is not always equivalent to double-spacing, since the current line may not be a double space line itself.	
² Skipping to a triple space line is not always equivalent to triple-spacing.	
³ Not all line printers have channels 9-12. The action taken for these devices will be as for CTL(3); i.e., single spacing.	
⁴ Use of this argument prints any pending characters and suppresses spacing; the next line will over- print the current line. Not all line printers have this capability; those that do not will single space.	
⁵ ASCII disc files cannot be logically rewound if the physical EOF has been encountered.	
⁶ A 12-inch leader and trailer is automatically punched. Should a program using a tape punch terminate abnormally (error, BREAK key, or disconnect), no trailer will be punched.	

CTL Function (Cont)

LINE PRINTER. The most commonly used CTL functions for the line printer are CTL(1) to skip to the top of form thereby starting printing on a new page, CTL(13) to suppress spacing and continue printing on the same line and the three line skip functions, CTL(14), CTL(15), and CTL(16). These last three functions allow true single, double, and triple spacing. This capability allows printing on each line of a page including over the perforations. Note that the 2767 line printer is physically incapable of supporting CTL(14), CTL(15), and CTL(16). If used for that printer, they will be converted automatically to CTL(3).

MAGNETIC TAPE. The control functions for the magnetic tape unit when used as an ASCII file are designed to facilitate use of both single file and multi-file reels of tape (but not multi-reel files). CTL(20) spaces forward until just prior to the first end-of-file mark encountered. If the tape was already so positioned, no action is taken. If the previous request was a write, an end-of-file mark is written and then backspaced over. Thus this control function always leaves the tape positioned at the end of the "current file", ready to extend it if so desired. Caution: if no end-of-file mark exists on the tape beyond the current position, no record exists beyond the end-of-tape mark, and the last request was not a write, then the unit will space over all of the tape and off the takeup reel. This condition cannot be diagnosed by the system and thus no diagnostic will appear (this problem can also occur with the next function described). CTL(21) spaces forward until just past the first end-of-file mark encountered (i.e., just prior to the first record, if any, of the next file). If the previous request was a write, an end-of-file mark is written and no further action is taken (i.e., the current file is completed and a new, empty file begun). If the end-of-tape mark has been passed when this request is made, the end-of-file action will be taken without any tape motion. CTL(22) backspaces until it just follows the first end-of-tape mark encountered (i.e., just prior to the first record of the current file), or to just following the load point indicator. If the previous request was a write, an end-of-file mark is written before backspacing. If the tape is already at the first record of a file, no action is taken. Thus this control function always leaves the tape positioned at the beginning of the current file. CTL(23) backspaces the tape until it just follows the second end-of-tape mark (or load point indicator) encountered (i.e., just prior to the first record of the preceding file). If the tape is currently positioned in the first file on the tape, the tape will be backspaced to the load point indicator and the end-of-file action is taken. If the previous request was a write, an end-of-file mark is written before backspacing. CTL(24) rewinds the tape to the load point indicator. If the previous request was a write, an end-of-file mark is written before rewinding.

Note that the system always ensures that an end-of-file mark appears after the last record written (unless that record crossed the end-of-tape mark, in which case the end-of-tape mark is treated like the end-of-file mark).

PAPER TAPE. The same CTL function arguments are used for output to and input from paper tape. In either case, the first three functions either punch or read ASCII strings. Following each string, an X-OFF, carriage return, and linefeed are punched as separators. When read, X-OFF, linefeed, NULL, and DEL are ignored; a carriage return delimits a record, control-H deletes the previous character, and control-X deletes the current line up to the next carriage return.

CTL Function (Cont)

If even or odd parity is requested by CTL(30) or CTL(31) respectively, a parity bit is punched on output and tested on input. If a character is read with incorrect parity, it causes the error message PRn-READ/WRITE FAILURE and the program terminates. The parity bit is stripped when the character is read. Note that even parity is the default when punching, parity ignored is the default when reading.

The fourth paper tape function CTL(33) simply punches or reads one 8-bit character per frame. Although leader and trailer frames are punched automatically, no inherent end-of-file condition exists in this mode. When reading such a tape, it is up to the user to test for the end of tape. One method is to punch two record lengths of null data (feed frames) and then when reading to check the data read for a record of all nulls. The record length is either the maximum for the device or that specified in the FILE command. This mode, sometimes called "transparent" is a good method for punching or reading binary data.

CARD READER. In the default mode CTL(40), Hollerith punch fields are automatically converted to their ASCII equivalents. Selecting CTL(42) converts Hollerith punch fields to their EBCDIC equivalents. The function CTL(41) allows cards to be read in column binary format. This means that the twelve punch positions are treated like one binary word of twelve bits. For each row punched a binary one is returned, and for each row not punched a binary zero is returned. Two characters are returned for each column read, the first character consisting of four bits corresponding to rows 12, 11, 0, and 1, and the second character consisting of the bits corresponding to rows 2 through 9 of that column. The following table shows the bits corresponding to the two characters. The first four bits of the first character are always zero.

Card Row	12	11	0	1	2	3	4	5	6	7	8	9
First Character	0	0	0	0	X	X	X	X				
Second Character							X	X	X	X	X	X

The X's represent the rows that may contain punches resulting in two 8-bit characters. For example, if rows 12 and 2 are both punched, the corresponding two characters are 00001000 and 10000000. In order to read column binary, the record length of the card reader must be specified as 80 words when the card reader is equated to an ASCII file. If fewer than 80 words are specified and if column binary is read from the file, characters at the end of the card may be lost. For example, if you specify a record length of 40 words, only the first 40 columns of the card will be read; any punches in the remaining 40 columns are lost.

When reading ASCII or EBCDIC data, you should specify a record length of 40 words when the card reader is equated to an ASCII file. This saves space in your program.

CTL Function (Cont)

READER/PUNCH/INTERPRETER. Use the CTL functions 40 through 54 to control how the card reader/punch/interpreter operates. All of the codes except CTL(43) set bits to establish how the device will operate on subsequent data transfers. These bits remain set (affecting both input and punch/print requests) until you change them or until your program terminates. Conversely, CTL(43) causes a one-time-only operation to immediately occur. CTL(43) is discussed later.

When not using CTL functions, the default condition for a read (input) request ejects any card in the visible wait station to stacker 1, picks a card from hopper 1, deposits the data card in the visible wait station, converts the Hollerith code to ASCII data, and transfers the data to your read variable. These events do not occur if your read request follows a punch/ print or CTL(43) operation. In this case, the device remains inoperative (both stacking and feeding do not occur for this one read operation) and the data transferred to your read variable is the data that was placed in the reader/punch/interpreter memory when a card passed through the read station during the previous punch/print or CTL(43) operation. When making read requests, you can use CTL(40), CTL(41), and CTL(42) in the same manner as previously described for a card reader. In addition, other control functions allow you to select the data mode, to assign hopper 2 for inputting cards, to assign stacker 2 for receiving cards, and to set the stacker overflow mode. In overflow mode, cards are stacked in stacker 1 until it is full, at which time stacker 2 is selected automatically.

When not using CTL functions, the default condition for a print (punch and/or interpret) request feeds a card from hopper 1, punches your ASCII data in Hollerith code, prints the same ASCII data at the top edge of the card, and then ejects the card to stacker 1. If there is a card in the visible wait station when you issue a print request, then that card is punched and/or printed. In either case, another card is next fed into the visible wait station from hopper 1. Generally, for consecutive punch operations, the cards being fed would be blank. Blank or not, however, as the card passes through the read station, the information it contains is stored in reader/punch/interpreter memory. Thus, if your next request is for input, the device remains inoperative and the information in device memory is transferred to your read variable.

When making print requests you can use control functions to change hopper and stacker assignments, select the data mode, and control punching and printing of the same or different data. A print request to punch column binary data, CTL(45), is the reverse of reading column binary (described under "Card Reader"). That is, for every two characters, the lower four bits of the first character are punched in rows 12, 11, 0, and 1 and the eight bits of the second character are duplicated in rows 2 through 9 of the column. Because two characters per column are punched, you must supply 80 words of binary data in order to punch all 80 columns of a card. If you plan to punch column binary data and print ASCII data on the same card, use CTL(41) to request column binary data mode, use CTL(53) to request punch and print separate data, and then supply 80 words of column binary data for punching followed by 40 words of ASCII data for printing. Note that a request to punch column binary implicitly selects the punch only mode.

Switching hopper and stacker assignments and controlling whether or not data is read or punched/printed becomes particularly important if you plan to interleave input and print requests in your program. Remember that except for CTL(43), once you set a control function it affects both input and punch/print operations until changed or until your program terminates. Thus, if you are reading data cards from hopper 1, the hopper assignment must be changed in order to punch blank cards from hopper 2. This is illustrated and explained in the example.

CTL(43) is a special one-time-operation function used to move the card currently in the visible wait station (if one is present) to a stacker without punching or printing and to move another card into the visible wait station without reading. This CTL(43) print request is generally

used, along with CTL(47) which selects hopper 2, to move a data card to a stacker and position a blank card in the visible wait station for the next punch/print request. It is interesting to note, however, that as the next card passes through the read station, the data it contains (including all blanks) is stored in reader/punch/interpreter memory — just as with cards that are fed during punch/print operations. Thus, if your next request is for input rather than for punch and/or print, the device remains inoperative (as described previously under read operations) and whatever the device memory contains is transferred to your read variable.

Example

The following program duplicates a deck of cards punched in ASCII format. The deck to be duplicated is placed in hopper 1 and the blank cards are placed in hopper 2. The original cards are ejected to stacker 1 and the new cards are ejected to stacker 2.

FILE-RPI,RP,40	Equate file RPI with Reader/Punch/Interpreter
10 FILES RPI	
20 DIM A\$(80)	
30 PRINT #1;CTL(51)	Set to punch only
40 IF END #1 THEN 100	Stop when ::card is read
50 LINPUT #1;A\$	
60 PRINT #1;CTL(48),CTL(47)	Select stacker 1 and hopper 2
70 PRINT #1;CTL(43)	Feed a card from hopper 2 and stack card in visible wait station in stacker 1
80 PRINT #1;CTL(49),CTL(46),A\$	Select stacker 2 and hopper 1. Punch the card and stack it in stacker 2. Feed a card from hopper 1.
90 GOTO 50	
100 PRINT "DUPLICATION DONE"	
110 END	

LINK TERMINAL. The control functions are designed to facilitate use of various capabilities offered by the link terminal.

CTL (70) is used to send HPIB messages to instruments connected locally to the link terminal via the HPIB. Messages are encoded in the print list following the CTL. The list of valid message formats is summarized in Appendix G.

CTL (71) enables you to switch indicator lights on or off. The print list following the CTL is interpreted as the number of an indicator light and that light is switched on if positive and off if negative. Valid arguments are -15 to -1 and 1 to 15. To illustrate:

```
PRINT #1;CTL(71),"- 1,2"
```

causes light #1 to go off and light #2 to go on.

CTL(72) enables you to define selected special function keys as terminators. When pressed during an input operation, a special function key that has been given the terminating mode, causes the input line to be immediately validated. The code generated by that special functions key appears as the last character of the line. The print list following the CTL is interpreted as the number of the key to be given the terminating mode (if positive) or the normal mode (if negative). Valid arguments are -10 to -2 and 2 to 10. To illustrate:

```
PRINT #1;CTL(72);+ 2,3,- 5
```

causes special function keys numbered 2 and 3 to become terminators and special function key 5 to become a normal data key.

CTL(73) is used to make the display unit print all numeric characters sent by the program, and echo all numeric characters entered through the keyboard. This is the default mode.

CTL(74) is used to prevent the display unit from printing characters sent to the link terminal by the program, and from echoing characters entered through the keyboard.

RJE ASCII FILES. The default control function for RJE devices, CTL(60), converts data as needed. For example when reading a JL or JP file sent from an IBM host function, EBCDIC code is converted to ASCII, and when printing to a JT device to be sent to an IBM host function, ASCII is converted to EBCDIC. Conversion is automatic with CDC, and conversion is not needed in 2000 to 2000 communication. A function, CTL(61), may be specified to prevent conversion of data when transmitting between an IBM host function and 2000. This might be used if the data received from an IBM host were simply to be stored and returned without processing.

DATA Statement

General Form:

DATA constant list

The DATA statement specifies data for READ statements.

A *constant list* is one or more *constants* separated by commas.

The data is read in sequence from first to last DATA statement, and from left to right within a given DATA statement.

DATA statements may be placed anywhere in a program except after the final END statement. The data items will be read in sequence as required by READ statements. The RUN and EXECUTE commands and a successful CHAIN statement reset the data pointer to the first item in the first DATA statement of the program.

The TYP function may be used to test the type of data items and the RESTORE statement may be used to move the data pointer without performing a read operation.

Examples:

```
10 DATA 457,"STRING DATA",2.192
20 DATA "AB" '65 '66 '7'10'13
```

See also: READ
TYP
RESTORE

DEF Statement

General Form:

DEF function name(parameter) = numeric expression

The DEF statement allows you to define special functions within a program.

A user defined function is one that is defined within the user program and is called within that program in the same way that one of the system-defined functions (i.e., SIN, SQR, TAN, etc.) is called.

The *function name* is made up of the letters FN followed by one of the letters A through Z (for example, FNC). The *parameter* has the form of a *numeric simple variable*. The *numeric simple variable* is a "dummy" variable whose purpose is to indicate where the actual argument of the function will be used when called. For example, in the following sequence, M is a dummy variable:

```
10 LET Y = 100
20 DEF FNA(M) = M/10
30 PRINT FNA(Y)
40 END
RUN
10
```

When FNA(Y) is called for in statement 30, the formula defined for FNA in statement 20 is used to determine the value printed.

Any operand in the program may be used in the defining expression; however, circular definitions such as:

```
10 DEF FNA(Y) = FNB(X)
20 DEF FNB(X) = FNA(Y)
```

cause a stack overflow error.

Examples:

```
60 DEF FNA(B2) = A**2 + (B2/C)
70 DEF FNB(B3) = 7*B3**2
80 DEF FNZ(X) = X/5
```

DIM Statement

General Form:

DIM dimension list

The DIM statement sets the amount of space allocated by the system for arrays and strings.

The *dimension list* is one or more dimension elements, separated by commas. A dimension element is a *string simple variable* followed by the string length in parentheses, an *array name* followed by the number of rows (one dimension array) in parentheses, or an *array name* followed by the number of rows and columns (two dimensional array) in parentheses.

DIM statements may occur at any point in the program. A variable may appear only once in DIM statements in a given program. The COM statement may be used instead of the DIM statement.

Example:

```
20 DIM A(5),B(12,7),G$(240),C(5,5),R0$(90)
```

END Statement

General Form:

END

The END statement terminates execution of the program and returns control to the system.

The END statement may occur at any point in the program and must be used as the last statement in all programs.

If a line printer, a punch, or other output device was assigned to a program, the last step taken at program termination is to verify that the output operation was successful. If a device error prevented successful completion of output, the device error must be corrected before the program will terminate. The BREAK key is disabled in this situation.

Example:

```
1000 END
```

ENTER Statement

General Form:

ENTER #*numeric variable*

or

ENTER [*#numeric variable*,] *time allowed*,*return variable*,*read variable*

The ENTER statement allows you to have greater control over data input than that available with the INPUT and LINPUT statements.

The ENTER statement may be used to limit the time allowed for data input, to test the actual response time of a given input, to determine the port number where input is entered, and to input data for one *read variable*.

If the number sign (#) and *numeric variable* are provided, the system stores the port number in the *numeric variable* as a value in the range 0 to 31.

The *time allowed* is a *numeric expression* which, when evaluated and rounded to an integer, specifies the time allowed for response in seconds. This expression should evaluate to a number (1 to 255). Zero is treated as one; numbers less than zero or greater than 255 are treated modulo 255 (256 = 0, 257 = 1, etc). Timing begins when all previous statements have been executed and any resultant output to the user terminal has been printed.

The *return variable* indicates the precise time in seconds the user took to respond. If the response is not acceptable, such as wrong data type, the value is the negative of the response time and the read variable remains unchanged. If the user failed to respond within the time limit, the value is set to -256. If a parity error occurred in transmission, -257 is returned. If a character was lost in transmission, -258 is returned.

The *read variable* is a *destination string* or a *numeric variable*. A character string is not enclosed in quotes, but may contain quotes and blanks. The extended literal string form ('71, '23, etc.) is not recognized. A carriage return alone is interpreted as a null string. If the *destination string* is not doubly subscripted and the character string entered is too long to fit in the *physical length* of the string, the *return variable* is set to -256 indicating an error, and data moved to the *destination string* is truncated to the *physical length* of the string. A *numeric variable* must be satisfied with a *numeric constant*.

The ENTER statement differs from the INPUT statement in that a "?" prompt is not displayed on the user terminal following data input and a linefeed is not generated following execution of the ENTER statement. The next program statement is executed whether or not the input time limit is exceeded.

Examples:

100 ENTER #V	(your port # is returned in V)
200 ENTER A,B,C\$	(you have A seconds to input C\$; B is the actual time taken)
300 ENTER #V,K1,K2,K3	(you have K1 seconds to input K3; V is set to your port # and K2 is the actual time taken)
400 ENTER 25,L,Q	(you have 25 seconds to input Q; L is the actual time taken)

EXP Function

General Form:

EXP (numeric expression)

The EXP function is a numeric-valued function which returns the mathematical constant "e" raised to the power of the *numeric expression* ($e^{\text{numeric expression}}$).

The approximate value of the constant e is 2.718282.

Example:

$$50 A = B * \text{EXP}(M/10)$$
FILES Statement

General Form:

FILES file list

The FILES statement opens files for use in a program.

The FILES statement does not create files (see the CREATE and FILE commands and the CREATE statement). The *file list* is made up of *file names* separated by commas. Up to four FILES statements can appear in a program, but only 16 files total can be declared (duplicate entries are allowed). The files are numbered (from 1 to 16) in the order they are declared in the program. These *file numbers* are used by READ, PRINT, LINPUT, ADVANCE, UPDATE, ASSIGN, and IF END statements, and TYP, REC, and ITM functions for file access and control.

A "*" may be used in the file list to reserve a position for a file to be named at a later point in the program using the ASSIGN statement. For example, 10 FILES AFILE, *, BFILE reserves file position number 2 for a file to be assigned later in the program. Note that a file name must be assigned to the reserved file number before a read, write, or test using the file number occurs, otherwise an error will result.

System library files may be accessed by using a "\$" in front of the file name (\$FRED). Group library files may be accessed by using a "*" in front of the file name (*TFILE). Files in other accounts may be accessed by adding the owner's idcode to the file name (file name.idcode) (HIS FIL.C901). Access to any files outside your own library are subject to that account's file access capabilities and restrictions.

If a file cannot be opened (the file does not exist or you are not allowed access), the program will terminate. If you are granted read-only access to the file because of its status or because it is in use, you will not be notified of this condition until you attempt to PRINT, which will cause program termination. Use of the ASSIGN statement (instead of FILES) to open the file will allow test of a *return variable* value to determine read/write access.

FILES Statement (Cont)

The system uses part of the user workspace for control information concerning the file approximately 18 words per *file name* position mentioned in the FILES statement. In addition, an area equal to the record length (1 to 1024 words) of the file is used in the user workspace as a buffer area for each file that is opened. A "*" used to reserve a position for a file in the FILES statement uses 256 words as a potential buffer. An ASCII disc file always uses 256 words regardless of the record length specification.

The same *file name* may be mentioned more than once in FILES statements used in a program to take advantage of the in-memory buffering of the file data described above (10 FILES AFILE,*,AFILE). During processing of the FILES statement, the checking the system normally performs to determine whether a file being opened is already in use is suspended (only for BASIC formatted files) for each occurrence of the same *file name* after the first reference to that *file name*. (The checking is not suspended if the file is open in another program.)

Files are opened when program execution begins, not when the FILES statement is encountered by the executing program. Thus, all FILES statements should refer to previously created files. If you desire to open a file created programatically, use the ASSIGN statement.

The syntax of a FILES statement is not carefully examined when it is entered in your program. Thus, if you include an invalid FILES statement you will not be informed by an error message until the program is executed.

Examples:

```
200 FILES MYFILE,$SYSFL,*GPFL,*,YOURFL.M350
300 FILES KEN,JIM,KEN,*,JIM
```

FOR and NEXT Statements

General Form:

FOR *for variable* = *initial value* **TO** *final value* [**STEP** *step size*]
NEXT *for variable*

The looping statements FOR and NEXT allow you to repeat a group of statements a specified number of times.

The FOR statement precedes the statements to be repeated, and the NEXT statement directly follows them. The number of times the statements are repeated is determined by the value of the *for variable*, which is a *numeric simple variable*. The *initial value*, *final value*, and the *step size* all are *numeric expressions* and are evaluated before the following steps occur.

When the FOR statement is executed, the *for variable* is set to the *initial value*. Then the following steps occur:

1. The value of the *for variable* is compared to the *final value*; if the *for variable* exceeds the *final value* (or if the *for variable* is less than the *final value* if the *step size* is negative), control skips to the statement following NEXT.
2. Statements between the FOR statement and the NEXT statement are executed in normal sequence order.
3. The *step size* (or 1 if the *step size* was not specified) is added to the value of the *for variable*.
4. Return to step 1.

Note that round-off errors may increase or decrease the number of steps (times through the loop) when non-integer step sizes are used. You should not execute the statements in a FOR loop except through a FOR statement. Transferring control into the middle of a loop can produce undesirable results.

Examples:

```

100 FOR P=1 TO 5
    .
    .
170 NEXT P
200 FOR R2=N TO X STEP -1.5      (where N≥X at start)
    .
    .
220 NEXT R2
250 FOR S=1 TO (Z*B) STEP (Y**2-V)
    .
    .
260 NEXT S

```


FOR and NEXT Statements (Cont)

Sample Program with a variable number of loops:

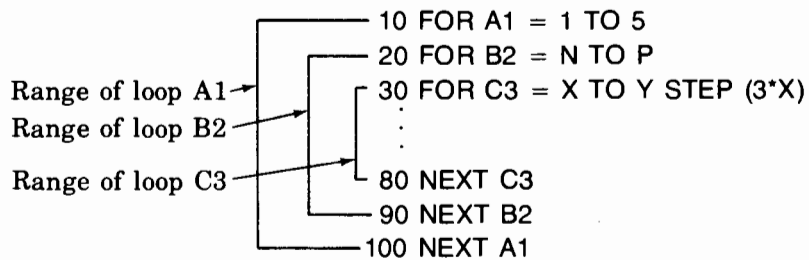
```

40 PRINT "HOW MANY TIMES DO YOU WANT TO LOOP";
50 INPUT A      (where A≤10)
60 FOR J = 1 TO A
70 PRINT "THIS IS LOOP";J
80 READ N1,N2,N3
90 PRINT "THESE DATA ITEMS WERE READ:"N1;N2;N3
100 PRINT "SUM = " ;N1+N2+N3
110 NEXT J
120 DATA 5,6,7,8,9,10,11,12
130 DATA 13,14,15,16,17,18,19,20,21
140 DATA 22,23,24,25,26,27,28,29,30
150 DATA 31,32,33,34
160 END

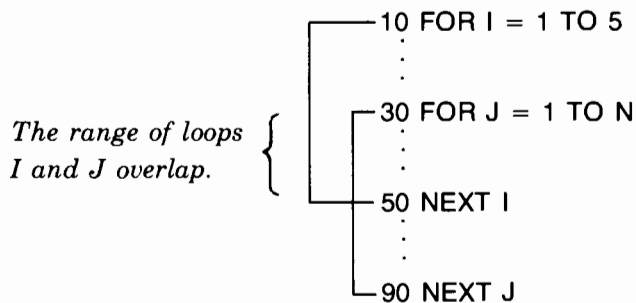
```

NESTING FOR . . . NEXT LOOPS. FOR . . . NEXT loops may be nested (placed inside one another) as long as the loops do not overlap.

Proper Nesting



Improper Nesting



GOSUB and RETURN Statements

General Form:

GOSUB *statement number*

or

GOSUB *numeric expression OF statement number list*
RETURN

The GOSUB statement overrides the normal sequential order of statement execution by transferring program control to the beginning of a subroutine.

A subroutine consists of a collection of statements that may be executed from more than one location in a program. In a subroutine, there is no explicit indication in the program as to which statements constitute the subroutine. The *statement number* to which control is transferred must be an existing statement in the current program. If the GOSUB transfers control to a statement that cannot be executed (such as REM, DIM, COM, DEF), control passes to the next sequential statement after the non-executable statement. A RETURN statement in the subroutine returns control to the statement following the GOSUB statement. There may be more than one RETURN statement in a subroutine.

Example:

```

50 READ A2
60 IF A2 > 100 THEN 80
70 GOSUB 400
.
.
380 STOP      (STOP frequently precedes the first statement of a subroutine to pre-
               vent accidental entry)
390 REM THIS SUBROUTINE ASKS FOR A 1 OR 0 REPLY
400 PRINT "A2 IS <= 100"
410 PRINT "DO YOU WANT TO CONTINUE?";
420 INPUT N
430 IF N # 0 THEN 450
440 LET A2 = 0
450 RETURN
.
.
600 END

```

GOSUB and RETURN Statements (Cont)

MULTIBRANCHING. Multibranch GOSUB statements use the value of a *numeric expression* to select the destination statements. The *numeric expression* is evaluated and rounded to an integer "n". Control is then transferred to the "nth" statement in the *statement number list*. The *statement number list* is one or more *statement numbers* separated by commas. Values of n less than 1 or greater than the number of branch statements in the *statement number list* are ignored.

Examples:

```
20 GOSUB 3 OF 100,200,300,400,500
60 GOSUB G-1 OF 200,210,220
70 GOSUB R OF 80,180,280,380,480,580
```

NESTING GOSUB STATEMENTS. Another subroutine can be called from within a subroutine. This is known as *nesting*. GOSUB statements may be nested logically to a level of 20. Nesting more than 20 GOSUB statements without an intervening RETURN statement will cause an error message. Note, however, that nested subroutines are exited in the reverse of the order in which they were entered (last in — first out). For example, if subroutine 250 (below) is entered from subroutine 200, 250 will be exited before subroutine 200.

Examples:

```
100 GOSUB 200
.
.
200 LET A = R2/7
210 IF A THEN 230
220 GOSUB 250
230
.
.
250 IF A > B THEN 270
260 RETURN to line 230
270 GOSUB 600
```

GO TO Statement

General Form:

GO TO *statement number*

or

GO TO *numeric expression* OF *statement number list*



The GO TO statement overrides the normal sequential order of statement execution by transferring program control to the specified *statement number*.

The *statement number* to which control is transferred must be an existing statement in the current program. If the GO TO statement transfers control to a statement that cannot be executed (such as REM, DIM, COM, DEF), control passes to the next sequential statement after the non-executable statement.

When the second form of the GO TO statement is used, the *numeric expression* is evaluated and rounded to an integer "n". Control then is transferred to the "nth" *statement number* in the *statement number list*, where *statement number list* is one or more *statement numbers* separated by commas. If there is no *statement number* corresponding to the value of the *numeric expression*, the GO TO statement is ignored and the statement following the GO TO statement is executed.

Extreme caution should be used if a GO TO statement is used to enter a FOR . . . NEXT loop. Doing so may produce an unpredictable result.

Examples:

```
10 GO TO X - Y OF 400,500,600
50 GOTO 100
80 GOTO 10
90 GOTO N OF 100,200,300
```

IDN Function

(refer to MAT)

IF . . . THEN Statement

General Form:

IF numeric relation THEN statement number

or

IF string relation THEN statement number

The IF . . . THEN statement transfers control to the specified *statement number* if the *numeric relation* or *string relation* is true. If the specified condition for transfer is not true, the statement following the IF . . . THEN statement will be executed.

A *numeric relation* is a *numeric expression*. If the *numeric expression* is evaluated to a non-zero number, it is considered "true"; it is considered "false" if evaluated to zero. Because numeric values may be rounded during computation, the "=" operator should be used carefully in IF . . . THEN statements. It is recommended that the "< =" or "> =" operators be used instead of "=" whenever practical when arithmetic computation is involved.

Example:

```
IF A < = B AND Z/2 > = M THEN 120
```

A *string relation* is composed of a *string variable* followed by a *relational operator* followed by a *string expression* (for example, A\$ = "ABC"). Strings are compared character by character within the two strings on either side of the *relational operator*. A given character is "less than" another character if the given character occurs in a lower position in the ASCII collating sequence (Appendix A) than the other character.

A character is "greater" if it occurs in a higher position in the ASCII collating sequence. A character is equal to another character only if they are the same character. (For example, "A" < "B" and "Z" > "1".) Note that all characters have a value (a position in the ASCII collating sequence) including blanks and non-printing characters.

If strings of unequal lengths are compared, and the characters in the shorter string are identical with the initial characters in the longer string, the shorter string is "less than" the longer string.

Two strings are "equal" only if they contain identical characters and are both of the same length.

Examples:

<pre>1. 5 DATA 2, 4, 6, 8, 10, 20, 30, 40 10 LET N=10 20 READ X 30 IF X<N THEN 60 40 PRINT "X IS 10 OR OVER" 50 GOTO 80 60 PRINT "X IS LESS THAN 10" 70 GOTO 20 80 END</pre>	<pre>2. CRE-AA, 64 5 FILES AA 7 DIM A\$(10) 10 INPUT A\$ 20 IF A\$="DONE" THEN 50 30 PRINT #1;A\$ 40 GOTO 10 50 END</pre>
---	---

```

3. 10 FILES AA
    20 IF END #1 THEN 80
    30 READ #1,R;V
    40 IF V>(R ** 2-1) OR R=3 THEN 80
    50 V=V ** 2
    60 LET R=R+1
    70 GOTO 30
    80 END

```

IF END Statement

General Form:

IF END # *file number* THEN *statement number*

The IF END statement causes a branch to a specified statement when an end-of-file (EOF) mark is detected during a file read operation; an attempt is made to write beyond the end of a file in a write operation; or a direct file write exceeds the record. When an EOF condition occurs the file is left positioned before the EOF mark. The system inhibits further reading of the file until a specific action clears the EOF. Such an action might be to rewind a magnetic tape or print to a disc file.

Since the IF END statement only sets a flag which stays set until another IF END statement is executed for the same file, or until the file is closed, it is not necessary to execute the statement prior to every file read or write.

Note that the branch is not taken if an EOF is encountered during execution of an ADVANCE statement.

If the IF END statement is not used and an EOF condition occurs during a file read or write operation, an error will occur terminating the program.

The statement will remain in effect throughout the program until changed by another IF END statement referencing the same file number. Note that the statement is associated with a *file number* and not a *file name*. The IF END trap is disarmed if the file is de-assigned.

The following example reads strings from a file until an EOF is reached, at which point the program informs the user that an EOF was encountered.

Example:

```

10 FILES FIL1
20 IF END #1 THEN 70
40 READ #1;A$
50 PRINT A$
60 GOTO 40
70 PRINT "END OF FILE"
80 END

```

IF ERROR Statement

General Form

IF ERROR THEN *statement number*

Execution of IF ERROR sets a flag in a program so that subsequent error messages are trapped. If an error (fatal, format, or file) occurs in any statement following IF ERROR, an automatic branch is taken to the specified *statement number* and the error number can be retrieved through the SYS function. If an error results only in a warning, no branch is taken but the error number can be retrieved through the SYS function.

Once specified, IF ERROR is disabled only by a CHAIN statement to another program, a STOP or an END statement. Note that the IF ERROR is disabled even if the CHAIN statement fails. The *statement number* to which control branches can be changed by a subsequent IF ERROR statement.

Execution of IF ERROR itself, is guaranteed not to engender an error. However, indeterminate results are possible since an error can be diagnosed in mid-statement. As a result of interrupting a program in mid-statement and transferring to another statement, such undesirable effects as uninitialized variables, partially updated files, and so forth, could occur.

The four types of errors that may occur (programming, format, warning, and file) are listed in Appendix C. Each error has a corresponding error number that is returned through the SYS(0) function when the IF ERROR trap is enabled. The SYS function returns other information that is useful when an error is trapped. (Refer to SYS function later in this section.)

An ASCII file error normally results in an error message to the user terminal and the system console. When such an error is trapped by IF ERROR, both of these messages are inhibited.

Example:

```
10 IF ERROR THEN 900
```

IMAGE Statement

General Form:

IMAGE *format string*

The IMAGE statement is used to represent the *format string* for a PRINT USING or MAT PRINT USING statement. See the PRINT USING statement discussion for a description of the contents of a *format string*. Note that a *format string* in an IMAGE statement is *not* enclosed in quotes.

The numeric equivalent string format (where '65 is equivalent to A) is not permitted in an IMAGE *format string*

The *format string* is not examined when you enter the IMAGE statement. Thus, you are not informed of an invalid format until the program is executed.

Example:

```
10 IMAGE # , 2(3DX/),"BOW-WOW",3AX
```

INPUT Statement

General Form:

INPUT read variable list

The INPUT statement allows you to input data from your terminal during program execution.

When the INPUT statement is executed, a "?" is displayed on the terminal and the program pauses until the input requirements are satisfied.

A *read variable list* is one or more *read variables* separated by commas. A *read variable* is a *numeric variable* or a *destination string*. For example, N, Y(3), A\$, and B0\$(6,7) are all *read variables*.

To respond to the "?" request for INPUT, you must type a list of *numeric constants* and/or *strings* separated by commas. The input data must be of the same type as the variables in the *read variable list*. If the data type does not agree with the variable type, you will receive the message "BAD INPUT, RETYPE FROM ITEM n," where n is the item number that was mistyped. Note that this item number refers to the line just typed. It does not refer to previous lines entered already in response to the same INPUT statement. If insufficient items are input, a "???" will be displayed on your terminal. If more items are input than were requested, you will receive the message "EXTRA INPUT — WARNING ONLY" and the additional items will be discarded. Should the system not properly receive the line typed, you will receive the message "TRANSMISSION ERROR. REENTER."

When entering a *numeric constant*, all characters before the first "+", "-", ".", or digit are ignored. Otherwise, the form of the number corresponds to the definition given under BASIC language elements.

When entering a *string*, leading blanks are ignored unless an opening quote is included. Trailing blanks are not ignored and a closing quote must be included unless the string is the last item entered on a line (or only item entered).

The only way to stop a program when input is required is to press the BREAK key. The program will stop unless the BRK function has disabled the BREAK key (refer to the BRK function). The program must be restarted using the RUN or EXECUTE command.

Examples:

```
10 INPUT A
20 INPUT A$,B,C(3)
30 INPUT P1$(N,N+20)
```


INT Function

General Form:

INT (*numeric expression*)

The INT function is a numeric-valued function which returns the greatest integer less than or equal to the *numeric expression*. For example, $\text{INT}(3.5) = 3$ but $\text{INT}(-3.5) = -4$.

Examples:

```
10 LET A = INT(X**Y)
20 PRINT INT(Z)
```

INV Function

(Refer to MAT . . . INV)

ITM Function

General Form:

ITM (numeric expression)

The ITM function returns the number of data items (numbers and strings) between the beginning of the currently accessed file record and the position of the file pointer for a file.

The *numeric expression*, when evaluated and rounded to an integer, is used as the *file number* to refer to the file (refer to the discussion of files in Section V).

ITM cannot be used with ASCII files; if attempted the program will be terminated with an error. Specification of an invalid *file number* will result in an error.

Example:

```
10 LET N=6
20 PRINT "I HAVE JUST READ ITEM" ITM(N) "FROM THE CURRENT RECORD OF FILE" N
```

LEN Function

General Form:

LEN (source string)

The LEN function returns the current (logical) length in characters for the specified string.

The DIM or COM statement specifies a maximum (physical) string length. The LEN function, however, allows you to check the actual number of characters currently assigned to a string variable. Do not confuse the LEN function with the LENGTH command.

Examples:

```
100 A = LEN(B$)
200 X$(LEN(X$)+1) = "ADDITIONAL SUBSTRING"
300 IF LEN(A$)#3 THEN 400
400 GOTO LEN(G$) OF 500,600,700,800
```

LET Statement

General Form:

[LET]replacement list = numeric expression

or

[LET]destination string = string expression

The LET statement assigns a value to one or more variables (10 LET B = 16.3).

The value assigned by the LET statement may be in the form of a *numeric expression* or a *string expression*. The *replacement list* is one or more *numeric variables* separated by replacement (assignment) operators ("="). In the LET statement, the equal sign ("=") is an assignment operator. It does not indicate equality, but is a signal that the value on the right of the assignment operator is to be assigned to the variable(s) on the left.

If a value is assigned to more than one variable, the assignment is made from right to left. For example, in the statement A = B = C = 2, first C is assigned the value 2, then B is assigned the current value of C, and, finally, A is assigned the value of B. Any *subscript* or *substring designator* qualifying a variable in the list is evaluated before assignment is made. For example,

```
5 N = 2
10 A(N) = N = 4           sets N and then A(2) equal to 4.
```

The rule that the equal sign ("=") is an assignment operator only holds as long as *numeric variables* occur in the *replacement list*. If a *numeric constant* appears, followed by an equal sign, that equal sign is treated as a relational operator. For example, in A=2=B the first "equal sign" (=) is treated as an assignment operator and the second is treated as a relational operator (that is, A is set to 1 or 0 depending on whether 2=B is true or false). Similarly, if a parenthesis appears, any subsequent equal sign is a relational operator. For example, A=(B=C) sets A to 1 or 0 depending on the value of the relational expression, B=C.

Note that the word LET is an optional part of the assignment statement.

```
10 LET A = 5.02
20 A = 5.02
30 X = Y7 = Z = Z(2) = 1
40 B$ = "ABC"
50 B$(1,5) = "ABCDE"
60 Z1$ = A1$(3,5)
70 M$(N,M) = A$(N,M)
```

LIN Function

General Form:

LIN (*numeric expression*)

The LIN function can be included in print operations to perform a carriage return and one or more line feed operations.

The LIN function can be used in PRINT, PRINT #, PRINT USING, PRINT# USING, MAT PRINT, MAT PRINT #, MAT PRINT USING, and MAT PRINT # USING statements. The *numeric expression* is evaluated and rounded to an integer. If the value is positive, the value of the expression specifies the number of line feeds to be generated. If the value is negative, the absolute value of the *numeric expression* will be used to determine the number of line feeds and the initial carriage return is suppressed.

Note that unless there is a trailing comma or semicolon at the end of the PRINT statement, the normal X-OFF, carriage return, and line feed characters will be generated in addition to those generated by the LIN function.

The LIN function is ignored in file print (PRINT # and MAT PRINT #) operations to BASIC formatted files but is allowed in print operations to ASCII files.

A comma following a LIN function in a print statement is treated like a semicolon (refer to the discussion of *print delimiter* under the PRINT statement).

Examples:

```
27 PRINT A, LIN(M*N/C1)
28 MAT PRINT C;LIN(10),D
29 PRINT M3,M(4,5),LIN(-A)
```

```
10 PRINT "ABC";LIN(-1);"DEF";LIN(2);"GHI"
20 END
RUN
```

```
ABC
  DEF
```

```
GHI
```

```
DONE
```

```
10 PRINT TAB(8);" TITLE:PRINT HEADING";SPA(10);"SUMMARY REPORT";
20 PRINT LIN(3);" DETAIL LINES"
30 END
RUN
```

```
TITLE:PRINT HEADING
```

```
SUMMARY REPORT
```

```
DETAIL LINES
```

```
DONE
```

LINPUT Statement

General Form:

LINPUT *destination string*

The LINPUT statement accepts an entire line of string data from the user terminal and assigns it to a *destination string*.

Unlike the INPUT statement, no prompt character (?) is printed when the LINPUT statement is executed.

All characters entered (including commas, quote marks, and leading and trailing blanks) are assigned to the string.

Examples:

```
30 LINPUT A$  
60 LINPUT X1$(10,20)
```

LINPUT # Statement

General Form:

LINPUT #*file number*;*destination string*

The LINPUT # statement reads the contents of the next available record into a *destination string*. The referenced file must be an ASCII file, or a user terminal. When *file number* is zero, the string data is expected from your terminal and LINPUT # is exactly equivalent to LINPUT.

Successive reads of the ASCII file cause successive records to be accessed. It is not possible to read the remainder of a record partially read by a preceding READ# statement.

Examples:

```
500 LINPUT # N+1; A$(6)  
510 LINPUT #5;A$  
520 LINPUT #1;B$(1,80)  
530 LINPUT #0;B$
```

LOCK Statement

General Form:

LOCK # *file number* [, *return variable*]

The LOCK statement is used to set or test a file status flag.

A file status flag is associated with each BASIC formatted file (refer to FILES Section V). The LOCK statement may be used to set the flag for a specific file (using the *file number*), indicating that you want exclusive access to the file. By using a *return variable* and testing its value, you can determine if the file is already locked. Similarly, once successfully "locked" by your program, other programs which check the status of the file will be alerted to the fact that the file is locked.

If no *return variable* is supplied, a program may only have one file locked at any given time. If a *return variable* is supplied, more than one file may be concurrently locked. If no *return variable* is supplied and the file is currently locked by some other program, your program will not resume execution until the previous program unlocks the file. If several files are locked and you do not supply a *return variable*, an error message is returned.

The file status flag is cleared by the UNLOCK statement, chaining to another program, or program termination. All pending write operations (from the locking program) are completed before the flag is cleared. The LOCK statement should always be used with a matching UNLOCK statement.

Note that the successful execution of a LOCK statement sets the file pointer at the beginning of the last accessed record in the file.

The meaning of the *return variable* for LOCK is as follows:

Return Value	Meaning
0	File locked successfully
1	File already locked (by your program or another program)
2	Invalid file number

Since the LOCK statement does not deny others access to the locked file, users must cooperate in its use for it to be effective. Locking is not necessary if more than one person does not access the file at the same time or if the file is not written to. For example:

```

20 FILE AFILE,BFILE
30 LOCK #1
40 READ #1;N
50 UNLOCK #1
60 LOCK #2,R
70 GOTO R + 1 OF 80,90,100
80 PRINT #2;N
85 UNLOCK #2
86 GOTO 30
90 PRINT "FILE BUSY"
95 GOTO 110
100 PRINT "FILE NUMBER ERROR"
110 STOP

```

LOG Function

General Form:

LOG (*numeric expression*)

LOG is a numeric valued function that returns the natural logarithm of the *numeric expression*, i.e., log to the base "e" ($\log_e(\textit{numeric expression})$).

The *numeric expression* must evaluate to greater than zero. If the evaluation results in zero, a warning message will be produced and the value of the log function is set to -10^{38} . If the argument is evaluated to a negative value (less than zero), the program terminates with an error.

Examples:

```
10 PRINT LOG(X)
20 A = LOG(B)
```

MAT Addition and Subtraction Statements

General Form:

MAT *array name* = *array name* + *array name*

or

MAT *array name* = *array name* - *array name*

The MAT addition statement sets an *array* equal to the sum of two *arrays* of the same dimensions. Addition is element by element ($A(I,J) = B(I,J) + C(I,J)$). The MAT subtraction statement sets an *array* equal to the difference of two *arrays* of the same dimensions. Subtraction is element by element ($A(I,J) = B(I,J) - C(I,J)$).

The dimensions of the resultant *array* must be the same as the component *arrays*. The same *array* may appear on both sides of the "=" sign.

Examples:

```
100 DIM A(20), B(20), C(20)
200 MAT A = B + C
300 MAT A = A + B
350 MAT A = A + A
400 MAT A = B - C
500 MAT A = A - B
```

MAT Assignment Statement

General Form:

MAT array name = array name

The MAT assignment statement is used to set one *array* equal to another array of the same dimensions. The transfer is element by element ($A(I,J) = B(I,J)$).

Example:

```
10 DIM A(10,20), B(10,20)
20 MAT B = A
```

MAT...CON Statement

General Form:

MAT array name = CON [new dimensions]

The MAT...CON statement sets up an *array* with all elements equal to one. The *new dimensions* parameter is optional.

Examples:

```
205 MAT C = CON
210 MAT A = CON(N,N)
220 MAT Z = CON(5,20)
230 MAT Y = CON(50)
```

MAT...IDN Statement

General Form:

MAT array name = IDN [new dimensions]

The MAT...IDN statement is used to establish an identity *array* (all 0's with a diagonal of all 1's).

Note that the *array* must contain the same number of rows and columns (must be square), or the optional *new dimensions* parameter must re-specify the dimensions as equal.

Sample identity matrix:

```
1    0    0
0    1    0
0    0    1
```

Examples:

```
205 MAT A = IDN
210 MAT B = IDN(3,3)
215 MAT Z = IDN(Q5,Q5)
220 MAT S = IDN(6,6)
```


MAT INPUT Statement

General Form:

MAT INPUT *array read list*

The MAT INPUT statement allows you to input entire *arrays* from the terminal.

An *array read list* is one or more *array names*, separated by commas. Each *array name* may optionally be followed by a *new dimensions* specification. The prompt characters and error messages used for the INPUT statement are used for the MAT INPUT statement.

Elements entered must be *numeric constants* (see the INPUT statement for special rules on entering *numeric constants*). Elements must be separated by commas and entered row by row, i.e., all of the elements in row 1 are entered before the elements in row 2, etc.

Note the difference between the MAT INPUT statement and the INPUT statement. For example, INPUT X(3,5) causes the fifth element of the third row to be input, while MAT INPUT X(3,5) causes the entire *array* X to be input, and sets its working size to 3 x 5.

Examples:

```
27 MAT INPUT A,B
30 MAT INPUT X,Z(15,N)
```

MAT...INV Statement

General Form:

MAT *array name* = INV(*array name*)

The MAT...INV statement is used to invert an *array*.

If *array* A is square and non-singular (determinant of A \neq 0), there exists a unique *array* (inverse of A) such that A times the inverse of A yields the identity matrix (see the MAT...IDN statement). For additional information on *array* inversion refer to the discussion of ARRAYS in Section III.

If you attempt to invert a singular *array* (determinant = 0), your program will terminate with an error.

Both *arrays* used in a MAT...INV statement must be square and be of the same dimensions. The same *array* may be used on both sides of the equation.

In performing the inversion, the system uses a temporary *array* requiring storage equal to the *array* being inverted. In some cases this may limit the size *array* that may be inverted.

Examples:

```
200 MAT A = INV(A)
300 MAT B = INV(A)
```

MAT Multiplication Statement

General Form:

MAT *array name* = *array name 1* * *array name 2*

Array multiplication sets an array equal to the product of two arrays.

The *array* which contains the final value must be large enough to hold the number of elements resulting from multiplying the number of rows in *array name 1* by the number of columns in *array name 2*. The number of columns in *array name 1* must be equal to the number of rows in *array name 2*. If array 1 is A(P,N) and array 2 is B(N,Q) then the resulting *array* would be C(P,Q) and would contain $P \times Q$ elements. For example, if A is multiplied by B:

MAT C = A * B

where the dimensions are A(3,2) and B(2,4), then C will have dimensions of (3,4). The same *array name* cannot appear on both sides of the equals sign.

Example:

200 MAT C = A * B

MAT PRINT Statement

General Form:

MAT PRINT *array print list* [*print delimiter*]

The MAT PRINT statement is used to print entire *arrays* in a single statement.

The *array print list* is any combination of *array names* and *print functions* separated by commas or semicolons (which are *print delimiters*). The *array print list* may be followed by an optional *print delimiter* (comma or semicolon), which controls output formatting.

A *print function* is one of the following:

- TAB (*numeric expression*) — tabs to column position
- LIN (*numeric expression*) — generates line feeds
- SPA (*numeric expression*) — spaces
- CTL (*numeric expression*) — outputs special device control commands

The elements of the *array* are printed row by row and can be spaced out or packed together (using commas or semicolons), as in the PRINT STATEMENT.

Each row of each *array* is printed separately, with double spacing between rows. If a comma follows the *array*, each element starts in one of the five divisions of the line. The output line is divided into five consecutive fields: four fields of 15 characters each and one field of 12 characters (starting at columns 0, 15, 30, 45, and 60) for a total of 72 characters. If a semicolon follows the *array*, the elements are printed packed together, as if each element were followed by a semicolon. If nothing follows the last *array*, a comma is assumed. All formatting is done according to the specifications under PRINT statement.

A one-dimensional *array* is printed as a single column.

Note that individual elements of an *array* can be printed using PRINT. For example,

```
100 PRINT A(1),A(2),C(50)
```

Examples:

Note the effect of the semicolons following A and B in the MAT PRINT statement, line 70, on the printed output. MAT READ in line 20 redimensions array B; redimensioning of arrays is not permitted in a MAT PRINT statement.

```
10 DIM A[10], B[5, 5], C[2, 2]
20 MAT READ A, B[3, 5], C
30 MAT PRINT A
40 PRINT
50 MAT PRINT B
60 PRINT
70 MAT PRINT A; B;
80 DATA 2.5, 46.7, 75, 0, 50.1, 0, 0, 0, 19.8, 0
90 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
100 DATA 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
110 END
```

MAT PRINT Statement (Cont)

RUN

2.5

46.7

75

0

50.1

0

0

0

19.8

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

2.5

46.7

75

0

50.1

0

0

0

19.8

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

DONE

MAT PRINT # Statement

General Form:

MAT PRINT # *file number* [, *record number*] ; END

or

MAT PRINT # *file number* [, *record number*] ; *array write list* [*print delimiter*] [END]

The MAT PRINT # statement prints *arrays* specified in the *array write list* on the specified file row by row. END which can be the last or only item in the *array write list* writes an end-of-file mark on the file. If the *file number* is zero, arrays are printed at your terminal exactly as if the MAT PRINT statement were used.

A serial print (no *record number* specified) prints an entire *array write list* on the file starting at the current position of the file pointer. A direct print prints the entire *array write list* starting at the beginning of the record specified in *record number*. An *array write list* is any combination of *array names* and *print functions* separated by, and optionally terminated by, *print delimiters*. In addition, the last or only item in the statement can be END, indicating that an end-of-file is to be written. A *print delimiter* is either a comma or a semicolon. Refer to PRINT # for a discussion of writing to files. Note that MAT PRINT # writes an entire *array* where PRINT # writes single elements.

Example:

The following example writes two *arrays* onto file BB. It then reads the *array* values into two different *arrays* with different dimensions.

```

10  FILES BB
20  DIM A(2,5), B(3,2)
30  MAT READ A
40  MAT READ B
50  DATA 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 1, 2, 3, 4, 5, 6
60  MAT PRINT #1; A, B
70  DIM C(5,2), D(2,3)
80  MAT READ #1, 1; C, D
90  MAT PRINT C, D
100 END
RUN

      10          20
      30          40
      50          60
      70          80
      90          100
      1           2           3
      4           5           6
DONE

```

MAT PRINT USING Statement

General Form:

MAT PRINT USING *format part* [; *array print list*]

The MAT PRINT USING statement allows you to control the output format of data from *arrays*. The *format part* is either a *format string* (represented as a *literal string* or *string variable*) or a *statement number* referencing an IMAGE statement. Refer to the PRINT USING statement for a description of the contents of a *format string*, noting that *string format specifications* are not allowed with MAT PRINT USING statements.

The optional *array print list* is any combination of *array names* and *print functions* (TAB, LIN, SPA, or CTL) separated by commas. Semicolons and trailing punctuation are not allowed. The commas serve as delimiters only; they have no formatting function. As with the MAT PRINT statement, the *arrays* are printed in row order.

If you include a meaningless *format string* in a MAT PRINT USING statement, you are not informed of an error until the statement is executed.

Examples:

```

9 IMAGE 10(DX)/
10 MAT PRINT USING 9; A, LIN(2)
20 MAT PRINT USING "S6D.2DXE/"; B,SPA(3),C
30 A$="3D.D"
40 MAT PRINT USING A$; A,B,C
50 A$='34"RIDICULOUS EXAMPLE" '34
60 MAT PRINT USING A$

```

MAT PRINT # USING Statement

General Form:

MAT PRINT # *file number*; USING *format part*; *array print list*

The file MAT PRINT USING statement allows you to format the printing of *arrays* to *ASCII files* as well as to a terminal. The file number specifies an open *ASCII file* or, if zero, your terminal. When used to print to a terminal, the statement is identical to the MAT PRINT USING statement.

The *format part* that defines format and the *array print list* that specifies the arrays to be printed are identical to those described for MAT PRINT USING.

If characters are sent to an *ASCII file* that extend beyond the end of the line (end of record), these characters are lost. Also overprinting provided by the carriage return character (+) or by the functions LIN(0) or CTL(13) will only work on *ASCII files* associated with a line printer capable of overprinting. For any other device, a new line is started when overprinting is requested.

Examples

```
10 MAT PRINT # 2;USING "S3D.DD";B
20 LET N=0
30 LET A$="3D.D"
40 MAT PRINT # N;USING A$;A,B
```

MAT READ Statement

General Form:

MAT READ *array read list*

The MAT READ statement reads entire *arrays* from DATA statements.

An *array read list* is one or more *array names* separated by commas. Each *array name* may optionally be followed by a *new dimensions* specification.

Example:

```
10 MAT READ A,B(10,12)
```

See also:

DATA

MAT READ # Statement

General Form:

MAT READ # *file number* [, *record number*] ;*array read list*

The MAT READ # statement reads array data from a file. If file number is zero, MAT READ # reads array data from DATA statements in the same way as a MAT READ statement.

If the optional *record number* is provided, reading begins with the data in that record. An *array read list* is one or more *array names*, separated by commas. Each *array name* may optionally be followed by a *new dimensions* specification.

When a file is accessed by record and more items are requested than exist in the record, an end-of-file (EOF) condition is generated. Serial file reads are limited only by the length of the file. If an ASCII file is referenced, data values are read in the same manner as in the INPUT statement.

Examples:

```
10 MAT READ #3;B
20 MAT READ #1,3;A(5,6)
30 MAT READ #5,N;A,B,M,Z
```

MAT Scalar Multiplication Statement

General Form:

MAT array name = (numeric expression) * array name

Scalar multiplication sets an *array* equal to the product of a number and an *array*.

Multiplication is element by element ($A(I,J) = \text{numeric expression} * B(I,J)$). Both *arrays* must have the same working size. The same *array* may appear on both sides of the "=" sign.

Examples:

```
100 MAT A = (5) * B
200 MAT A = (N/3) * A
300 MAT A = (Q7 * N5) * C
```

MAT...TRN Statement

General Form:

MAT array name = TRN(array name)

The MAT...TRN statement is used to set an *array* to the transposition of a specified *array*.

Transposition causes rows and columns to be switched (row 1 becomes column 1, etc). The same *array* cannot be used on both sides of the "=" sign.

Sample transposition:

Original Array

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

Transposed Array

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

Note that the dimensions of the resulting *array* must be the reverse of the original *array*. For example, if A has dimensions of 6,5, and MAT C = TRN(A), C must have dimensions of 5,6.

Example:

```
300 MAT A = TRN(B)
```


MAT...ZER Statement

General Form:

MAT *array name* = ZER [*new dimensions*]

The MAT...ZER statement sets all elements of the specified *array* to zero. The *new dimensions* parameter is optional.

Examples:

```
305 MAT A = ZER
310 MAT Z = ZER(N)
315 MAT X = ZER(30,10)
320 MAT R = ZER(N,P)
```

NEXT Statement

(Refer to FOR)



NUM Function

General Form:

NUM(*source string*)

The NUM function returns the numeric ASCII code value of the first character in the specified *source string*. For example, NUM("A") would return 65. NUM of a null (zero-length) string returns 0.

A list of ASCII character values is given in Appendix A.

Examples:

```
200 PRINT NUM(A$(J,J))
300 GOTO NUM(X$(LEN(X$))) - 60 OF 400,500,600
```

See also:

CHR\$

POS Function

General Form:

POS(*source string 1*, *source string 2*)

The POS function returns the character position in *source string 1* where the first incidence of *source string 2* occurs. For example, POS("12ABC34","ABC") would return 3.

If *source string 2* is not a substring of *source string 1*, then the value returned is 0. If *source string 2* is a null string, then the value returned is 1.

Examples:

```
200 PRINT POS(A$,B$(3,7))
300 A$(J) = B$(POS(B$,G$),POS(B$,G$) + LEN(G$))
```

PRINT Statement

General Form:

PRINT

or

PRINT *print list* [*print delimiter*]

The PRINT statement causes data to be output at the terminal.

The data to be output is specified in a *print list*. A *print list* consists of items (*numeric expressions*, *string expressions*, and *print functions*) separated by commas or semicolons (which are *print delimiters*). The *print list* may be followed by an optional *print delimiter* (comma or semicolon) which controls output formatting. If the *print list* is omitted, PRINT causes a skip to the next line.

A *print function* is one of the following:

- TAB (*numeric expression*) — tabs to column position
- LIN (*numeric expression*) — generates line feeds
- SPA (*numeric expression*) — spaces
- CTL (*numeric expression*) — outputs special device control commands

The contents of the *print list* are printed. If there is more than one item in the *print list*, commas or semicolons must separate the items. The choice of a comma or semicolon affects the output format.

The *print delimiter* between items can be omitted if one or both of the items is a *literal string*. In this case, a semicolon is inserted automatically. For example PRINT "ABC" "DEF" would be printed in the same format as PRINT "ABC"; "DEF".

The output line is divided into five consecutive fields: four fields of 15 characters each and one field of 12 characters, for a total of 72 characters. When a comma separates items, each item is printed starting at the beginning of a field. When a semicolon separates items, each item is printed immediately following the preceding item. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line.

An X-OFF, carriage return, and line feed are output after each PRINT has executed, unless the output list is terminated by a comma or semicolon. In this case, the next PRINT statement begins on the same line.

If a *numeric expression* appears in the *print list*, it is evaluated and the result is printed. Any variable must have been assigned a value before it is printed. A *literal string* is output "as is".

Numeric values are left justified in a field whose width is determined by the magnitude of the number. The smallest field is six characters. Numeric output format is discussed in detail below.

PRINT Statement (Cont)

Examples:

When items are separated by commas, they are printed in up to five fields per line; separated by semicolons, they directly follow one another. In the following example, the items are numeric, so each item is assigned a minimum of six print positions.

```

10 LET A=B=C=D=E=15
20 LET A1=B1=C1=D1=E1=20
30 PRINT A, B, C1, C
40 PRINT A;B;C1;C;D;E;E;A1;D1;E1
50 PRINT A, B; C, D
60 END
RUN

```

```

15          15          20          15          20          15
15      15      20      15      15      15      15      15      20      20      20
15          15      15          15

```

DONE

In the next example, a DIM statement is used to specify the number of characters in each string.

```

10 DIM B$(3), C$(3)
20 C$="ABC"
25 B$="ABC"
30 PRINT B$, C$
40 END
RUN

```

```

ABC          ABC

```

DONE

In the example below, the first PRINT statement evaluates and then prints three *numeric expressions*. The second PRINT skips a line. The third and fourth PRINT statements combine a *literal string* with a *numeric expression*. No fields are used in the print line for *literal string* unless a comma appears as a separator.

```

10 LET A=B=C=D=E=15
20 LET A1=B1=C1=D1=E1=20
30 PRINT A*B, B/C/D1+30, A+B
40 PRINT
50 PRINT "A*B ="; A*B
60 PRINT "THE SUM OF A AND B IS"; A+B
70 END
RUN

```

```

225          30.05          30

```

```

A*B = 225
THE SUM OF A AND B IS 30

```

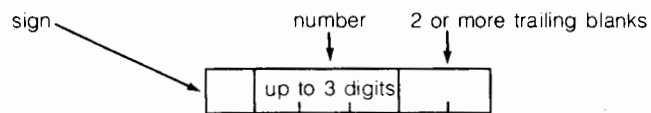
DONE

PRINT Statement (Cont)**NUMERIC OUTPUT FORMATS**

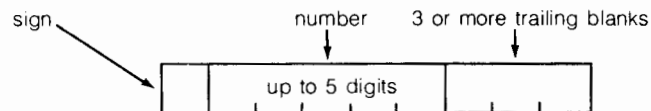
This discussion defines the output format of *numeric expressions* that are followed by a semicolon or an implied semicolon.

Numeric quantities are left justified in a field whose width is determined by the magnitude of the item. The width includes a position at the left of the number for a possible sign and at least one position to the right containing blanks. The width is always a multiple of three; the minimum width is six characters. If a numeric quantity is negative, a minus sign is printed; if it is positive, a blank is printed.

INTEGERS. An integer with a magnitude less than 1000 requires a field width of six characters:



An integer with a magnitude between 1000 and 32767 inclusive requires a field width of nine characters:



Examples of integers:

The integers below are less than 1000 and greater than -1000:

```
10 PRINT 1;999;30;-300;+295
20 END
RUN
```

```
1      999    30   -300   295
```

```
DONE
```

These integers are between 1000 and 32767 or between -1000 and -32767.

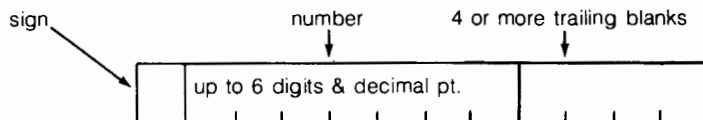
```
10 PRINT 1000;+32751;-32767;25678
20 END
RUN
```

```
1000    32751   -32767   25678
```

```
DONE
```

PRINT Statement (Cont)

FIXED-POINT NUMBERS AND ALL OTHER INTEGERS. Fixed-point numbers and all other integers require a field width of 12 positions. If the magnitude of the number is greater than or equal to .09999995 and less than 999999.5, or is less than .1 but can be printed with six significant digits, the number is printed as a fixed-point number with a sign. Trailing zeros are not printed, but a decimal point is printed. The number is left-justified in the field with trailing blanks. The sign is printed only if it is negative.



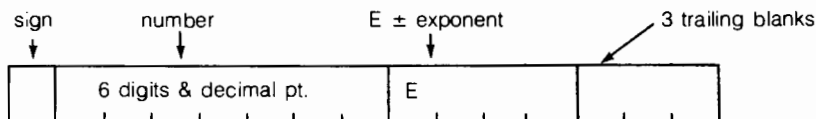
Examples of fixed-point numbers:

```
10 PRINT 9.99999E+05; .1; .000044
20 END
RUN
```

```
9.99999E+05      .1      .000044
```

DONE

ALL OTHER NUMBERS. Any number, integer or fixed-point, with a magnitude outside the range of the magnitude of the numbers presented above, is printed as a floating-point number using a total field width of 15 positions:



Examples of floating-point numbers:

```
10 PRINT 2.34568E+06; 4.4E-06
20 END
RUN
```

```
2.34568E+06      4.40000E-06
```

```
10 PRINT 2.34568E+07; 4.4E-07
20 END
RUN
```

```
2.34568E+07      4.40000E-07
```

```
10 PRINT 3.943E-05; 2.57895E-05
20 END
RUN
```

```
3.94300E-05      2.57895E-05
```

PRINT # Statement**General Form:**

PRINT # *file number* [, *record number*]

or

PRINT # *file number* [, *record number*] ; END

or

PRINT # *file number* [, *record number*] ; *write list* [*print delimiter* [END]]

The PRINT # statement writes data and control information to BASIC formatted and ASCII files, or to your terminal.

When *file number* evaluates to an integer greater than zero, it specifies the file to which data is written; when it is zero, the PRINT # statement is equivalent to the PRINT statement and prints data serially at your terminal. The *record number* parameter may not be used when *file number* is an ASCII file or zero.

A *write list* is any combination of *numeric expressions*, *string expressions* and *print functions* separated by *print delimiters*. A *print delimiter* is a comma or a semicolon. In addition the last or only item in a *statement* can be the special print function: END. END and any CTL functions are ignored when *file number* is zero. If the *write list*, *record number*, and END are all omitted, no write is performed on BASIC formatted files; the current line is completed on ASCII files.

BASIC FORMATTED FILE PRINTS. Serial (Sequential) Access Printing. Writing data to a file without specifying a *record number* causes the file to be filled serially, without respect to record boundaries. Successive prints cause data items to be stored one after the other beginning at the current position of the file pointer. A serial file print leaves the file pointer positioned immediately following the last item in the *write list*. Data written serially is usually read serially using successive file read (READ #) operations. Specifying END causes an end-of-file (EOF) mark to be written on the file following the last item in the *write list*. One word is used for the EOF unless no more space remains in the record or in the file, in which case the end of the record or file constitutes the EOF. Writing an END leaves the file pointer positioned before the END.

Examples:

```
10 PRINT #1; A$,B1,N$(1,2)
20 PRINT #N+1; "TEXT",B$,X*Y,END
```

Direct (Random) Access Printing. Use of the *record number* after the *file number* in a PRINT # statement allows data to be stored in the record specified by the *record number*. Each direct file print causes items in the *write list* to be written at the beginning of the selected record. A direct file print leaves the file pointer positioned in the record immediately following the last item in the *write list*. Writing an END leaves the file pointer positioned before the END.

In a direct file print operation, if the data in the *write list* exceeds the amount of space available in the record, an end-of-file (EOF) condition is generated. In both direct and serial

PRINT # Statement (Cont)

access, if printing beyond the physical end of the file is attempted, an end-of-file (EOF) condition will occur. A trailing *print delimiter* (comma or semicolon) has no meaning for serial or direct access to BASIC formatted files. Print functions (TAB, LIN, SPA, and CTL) are ignored for serial or direct access to BASIC formatted files.

Examples:

```
10 PRINT #1,3;A$,B$,F2,E(3,4)
20 PRINT #N,R(3); A,B,"TEXT",END
```

Serial and direct file print statements can be used to write on the same file. A serial print following a direct print will write data immediately following the previous data.

The first record of a file is record number 1.

Numeric items occupy two words in a file. Strings occupy one word specifying the current *logical length* and one word for each two characters in the string. A string with an odd number of characters uses one word for the last character.

Serial File Print Example:

```
10 FILES AFILE
20 DIM A$(5)
30 READ A,B,C,D,E,F,G,H,I,J
40 DATA 100,200,300,400,500,600,700,800,900,1000
50 LET A$="ABCDE"
60 PRINT #1;"NUMBERS",A,B,C,D,E,F,G,H,I,J,END
80 PRINT #1; A$
90 PRINT #1;END
100 END
```

The string "NUMBERS" and the numbers 100 through 1000 are written onto the file number 1 (AFILE). An end-of-file mark is written following the last data item. Line 80 overlays the end-of-file mark previously written on AFILE with the string "ABCDE". Since the END is omitted, an end-of-record mark is automatically written after the string. Line 90 replaces the end-of-record mark with an end-of-file mark.

Direct File Print Example:

```
10 FILES AFILE
20 READ A1,B1,C1,D1,E1
30 DATA 1,2,3,4,5
40 LET A$="A"
50 FOR N=1 to 10
60 LET B(N)=N+1
70 NEXT N
80 DIM B(10)
90 PRINT #1,1;"START OF AFILE"
100 PRINT #1,2; 10,A1,B(1),B(2),B(3)
110 REM. TWO RECORDS HAVE BEEN WRITTEN ON AFILE
120 PRINT #1,2; B1,C1,D1,E1
```


PRINT # Statement (Cont)

```
130 PRINT #1,1;A$
140 PRINT #1,3;END
150 REM. .THE THIRD RECORD OF AFILE IS AN END-OF-FILE
160 END
```

The first record of file number 1 contains a string value. The second record has five numeric items. The program writes four numeric items in the second record (overlying the previous five items). A different string is written on record 1 and an end-of-file is written on record 3. Note that the records do not have to be written in the order they appear in the file.

ASCII FILE PRINTS. Printing to an ASCII file causes the data to be formatted in the same way as a print to your terminal. Numeric and string data items are written one per field (where a field is 15 print positions wide) if the items are separated with commas; they are written closely packed if separated with semicolons. The record length of an ASCII file determines the number of print fields per record. A print that generates a line longer than the record size of an ASCII file will result in truncation of the line by removing excess characters from the end of the line. Each print to an ASCII file results in a new line unless a trailing *print delimiter* exists in the *write list*.

A record number cannot be specified when printing to ASCII files, but all of the *print functions* (TAB, LIN, SPA, and CTL) can be used and have the same effect as in a normal print to a terminal. Note that PRINT statements do not automatically print commas between *write list* items, and that ASCII file read operations require commas between data items (the same as if the data was input from your terminal). To meet this requirement, include a comma as a *literal string* (“,”) between items in the *write list* if the file is to be read later using a file read statement. Alternately, you can use the LINPUT # and CONVERT statements to read the records and extract numeric values from them.

Device status messages are sent to both your terminal and the system console to report device malfunctions or conditions requiring operator action. The messages DEVICE NOT READY, DEVICE ERROR, and ATTENTION NEEDED report device-specific conditions which cause suspension of the user operation. In the first two cases the system automatically restarts the operation after corrective action has been taken. For the ATTENTION NEEDED message the operator must use the AWAKE command to resume the user operation. The message READ/ WRITE FAILURE is printed for non-recoverable errors. In this case the user operation is terminated in the same manner as a program error.

Refer to the description of ASCII files in Section X for details of particular devices used as ASCII files.

PRINT USING Statement

General Form:

PRINT USING *format part* [; *using list*]

The PRINT USING statement gives you more control over the format of your output data than the PRINT statement, but requires additional programming effort. The *format part* is either a *format string* (represented as a *literal string* or *string variable*) or a *statement number* referencing an IMAGE statement.

The optional *using list* specifies the items to be printed. It is any combination of *numeric expressions*, *string variables*, or *print functions* (TAB, LIN, SPA, or CTL) separated by commas. Semicolons and trailing punctuation are not allowed. Commas in the *using list* serve as delimiters only; they have no formatting function. Note that *string variables*, not *string expressions*, are allowed. The *using list* may be omitted only when the *format part* does not contain any D or A specifications.

A *format string* consists of an optional *carriage control* character and comma followed by one or more *format specifications* or *groups of format specifications* separated by a comma and/or slashes). Each *format specification* is either a *literal string* or an orderly combination of *format characters*. *Format strings* may be up to 255 characters long. Blanks are ignored and lower-case characters are upshifted, except within literal strings.

Example:

```
100 PRINT USING "S4D.DD,4A"; B,A$, LIN(1)
```

CARRIAGE CONTROL CHARACTERS. One of the following optional *carriage control* characters may appear as the first non-blank character of a *format string*:

- + suppress linefeed
- suppress X-OFF and carriage return
- # suppress X-OFF carriage return, and linefeed

If supplied, the *carriage control* character must be followed by a comma and at least one slash, *format specification*, or *group*. The specified action occurs at the completion of the PRINT USING statement. If the *carriage control* character is not supplied, the default action is to terminate the line with X-OFF, carriage return and linefeed.

FORMAT CHARACTERS. *Format specifications* are comprised of orderly combinations of the following *format characters*:

- A reserves one character position in a string specification
- D reserves one decimal digit position in a numeric specification
- S defines the position of the sign character in a numeric specification
- . defines the position of the decimal point in a numeric specification
- E specifies floating point format in a numeric specification
- X reserves one blank character position in a numeric, string, or literal specification.

An optional *repetition factor* (replicator), between 1 and 255 inclusive, may precede an A, X, or D (e.g. 3A is equivalent to AAA).

PRINT USING Statement (Cont)

DELIMITERS. *Format specifications* must be separated with either a comma or a slash (/). A comma serves only as a delimiter whereas a slash also causes an X-OFF, carriage return, and linefeed to be output. Commas may appear only between *format specifications* and may not be adjacent. Slashes may appear before, after, and in place of *format specifications* as well as between them, therefore slashes may be adjacent to commas. Multiple slashes (///) are allowed but a slash may not be preceded by a *repetition factor*.

GROUPS. A *group* of one or more *format specifications* may be enclosed in parentheses which must be preceded by a *repetition factor* between 1 and 255 inclusive (e.g. 2(/AX,D/) is equivalent to /AX,D//AX,D/). Within the parentheses, the specifications must be separated by commas or slashes and the *group must be set off from other specifications by a comma or slashes, just as if it were a single specification. Groups can be nested two levels deep.*

REPETITION FACTORS (REPLICATORS). An unsigned positive integer between 1 and 255 inclusive must precede the left parenthesis (of a *group* and may optionally precede any X, D, or A in a *format specification*. It indicates the number of times the following character or group is to be repeated. Leading zeros are allowed.

EXECUTION OF THE PRINT USING STATEMENT. Execution of the PRINT USING statement commences by examining the *format string*. The *carriage control* character, if present, is noted for termination processing, then each *format specification* is examined.

If the specification is either a string or a numeric specification, the next item from the *using list* is printed according to the specification. If the *using list* has already been exhausted or is not present, the statement terminates. If the item does not agree with the specification (i.e. string vs. numeric), an error message is printed and the program execution terminates.

If the specification is literal, the specified number of blanks (or the contents of the literal string) is simply printed; the *using list* is not examined.

If the end of the *format string* is reached before the end of the *using list*, processing continues from the beginning of the *format string* but after the optional *carriage control* character (if the *format string* contains no string or numeric specifications, the statement terminates).

When all items from the *using list* have been printed the statement terminates (any remaining literal string specifications are processed if the end of the *format string* has not been reached for the first time). Termination consists of printing an X-OFF, carriage return, and linefeed, modified by the *carriage control* character.

If the *format string* is empty or contains only blanks, output consists of only an X-OFF, carriage return, and linefeed.

FORMAT SPECIFICATIONS. There are three categories of *format specifications*: those containing at least one D (numeric), those containing at least one A (string), and those consisting of either a literal string or all X's (literal); A's and D's may not appear in the same *format specification*. Numeric specifications are further classified as integer (no decimal point or exponent), fixed-point (a decimal point but no exponent), and floating-point (an exponent and possibly a decimal point). Table 10-4 describes the output formats that result from each type of *format specification*.

Format Specification Examples

TYPE	COMBINATION RULES	EXAMPLES		
		FORMAT SPEC	VALUE	OUTPUT
<p>INTEGER</p>	<p>Any combination of X's and D's is allowed, but at least one D must be present and only one S is allowed. This specification must match a number in the print list. The number is rounded to an integer and printed right-justified. Although the requested number of digits will be printed, only six are significant.</p> <p>If there is not enough room in the field for the number (i.e. the number of digits is greater than the number of D's in the format), then the value is printed on a separate line in a floating point format (SD.5DE).</p> <p>If an S precedes all D's, the sign is printed immediately preceding the first digit of the number. If an S appears after the first D, the sign is printed at the location of the S.</p> <p>If an S is not included in the format, then an extra D should be provided if the value is negative. When the value is negative, the - sign always precedes the most significant digit and a space must be provided with an extra D to prevent overflow.</p> <p>Blanks may be combined with carriage control to cause overprinting. For example, large numbers can be printed with blanks in the positions reserved for commas (e.g. \$10,937).</p>	<p>DDDD } 4D } equivalent 2DDD } 2D2D } 2(2D) }</p> <p>4D 1234.8 1235 S4D 1234 +1234 4DS 1234 1234+ 5D 1234 ^1234 5D -1234 -1234 DX3D 1234 1^234 DS3D 1234 1+234 S10D 1234 ^^^^^^+1234</p>		
<p>FIXED-POINT</p>	<p>Any combination of D's and X's to the left and right of the decimal point is allowed, but at least one D must be present and only one S and one "." are allowed. For this specification, the next item in the print list must be a numeric quantity. The digits to the right of the decimal point are rounded to fit the field. Leading zeros to the left are suppressed, but trailing zeros are printed.</p> <p>If the number to be printed has no digits to the left of the decimal point but D's are provided to the left, then a zero will be printed in the rightmost D on the left side. If an S is provided to the left, it is moved to the right through D's and X's until it comes to the first non-blank character. If an S is not provided and the number is negative,</p> <ol style="list-style-type: none"> no D's to the left causes overflow; one D to the left will be used for the minus sign and the leading zero will be dropped; or two or more D's to the left will have the minus sign and zero printed in the two rightmost D positions. 	<p>DDD.DDDD } DDD.4D } equivalent 3D.4D } 3D.DDDD }</p> <p>465.465 ^465.4650 4D.2D 465.465 ^465.47 4D.3D -465.465 -465.465 SDD2D.D 465.465 ^+465.5 S2D.4D .465 ^+0.4650 S.4D .465 +.4650 D.4D -.465 -.4650 2D.4D -.465 -0.4650</p>		

Format Specification Examples (Continued)

TYPE	COMBINATION RULES	EXAMPLES		
		FORMAT SPEC	VALUE	OUTPUT
<p>FLOATING-POINT</p> <p>Any legal INTEGER or FIXED-POINT format specification may be followed by an E. The E signifies a four character field consisting of an "E" followed by "+" or "-" and two decimal digits. This format is useful for numbers that are very large or very small. For example, .00005 = $.5 \times 10^{-4} = .5E-4$. When X's follow the E they cause blanks to be printed between the E and the exponent sign.</p> <p>The output value from the print list is multiplied or divided by 10, the number of times necessary to fit the value into the field. It is then rounded from the right, and the exponent is adjusted to match the number of multiplications or divisions.</p> <p>If the format allows for more digits than there are significant digits in the output value, two rules are followed:</p> <ol style="list-style-type: none"> 1. If there are more than 6 D's on the right side of the decimal point, the leftmost digit is printed in the first D (if any) to the left of the decimal point or the first D to the right of the decimal point; extra D's beyond 7 on the right are filled with non-significant digits. In the following examples, the arrow indicates the position of the leftmost digit printed: DD.40D XX.DD40D 40DD.40D ↑ ↑ ↑ 2. If there are less than 7 D's on the right side of the decimal point, the leftmost digit is printed in the seventh D position from the right (or the leftmost if there are not 7). In the following examples, the arrow indicates the position of the leftmost digit printed: 6DDDD.DDDD DD.DD D.6D ↑ ↑ ↑ 		<p>SDXE DDDD.DDE S5DX.X5DEX SD.5DE S.10DE3X</p>	<p>4.82716X10²¹ ↓</p>	<p>+5^E+21 4827.16E+18 ^^^+48^..27159E^+20 +4.82716E+21 +.4827159382E^^^+22</p>
<p>STRINGS</p> <p>Any combination of A's and X's. Strings are left-justified in the field. Leftover spaces are filled with blanks. If the string contains more characters than the specification allows, characters are truncated from the right.</p>		<p>AAAA } 2A2A } equivalent 2(2A) } 4A } 6A } 8A } 2X6A } AXAXAXAXAXA }</p>	<p>ABCDEF ↓</p>	<p>ABCD ABCDEF ABCDEF^^ ^^ABCDEF A^B^C^D^E^F</p>
<p>LITERAL</p> <p>A specification consisting only of X's or any literal string.</p> <p>If the format string is represented as a literal string and contains literal string specifications, the outer quote marks must be represented by using the extend string literal convention (i.e. '34).</p>		<p>XX4X Literal^string 1</p>	<p>none none</p>	<p>^^^^^^ Value of Literal String</p>

PRINT USING Statement (Cont)**Print Using Format Errors****FATAL ERRORS**

These errors cause termination of the program. An error message is printed along with the offending specification.

1. Replicator is outside the range $1 \leq n \leq 255$.
2. D, S, E, or . occurs in a string specification.
3. A occurs in a numeric specification.
4. Characters other than A, X, D, S, E, /, or . occur in any specification other than a literal.
5. Adjacent commas.
6. More than two levels of parentheses.
7. No D in a numeric specification.
8. An S in a blank specification.
9. String expression encountered for a numeric format specification.
10. Two or more E's, S's, or .'s in a specification.
11. Replicator not followed by (, D, X, or A.
12. Literal string not set off by delimiters and enclosed in quotes.
13. Parentheses used without a replicator.
14. Referenced statement is not IMAGE.
15. Numeric data encountered by a string format.
16. Leading or trailing commas in group or format string (i.e. null group or missing format specification).

NON-FATAL ERRORS

These errors do not terminate the program.

1. String specification too small for the string data causes the string to be truncated from the right.
2. Field too small for number. Causes the number to be printed on the next line using SD.5DE format. Printing resumes on the next line.

PRINT USING Example:

```

10 LET B=12345
20 LET AS="ABCDEFGHJ"
30 DIM AS(10)
110 PRINT USING "S4D.DD,4A";B,AS(4,8),LIN(1)
120 PRINT USING 125;AS
125 IMAGE "PRINT 3 LETTERS",XXAAA
130 PRINT USING '34"END OF PRINT EXAMPLE"'34
210 END
RUN

```

```

+1.23450E+04
DEFG

```

```

PRINT 3 LETTERS ABC
END OF PRINT EXAMPLE

```

```

DONE

```

PRINT # USING Statement

General Form:

PRINT # *file number*; USING *format part* [*using list*]

The file PRINT USING statement allows you to format output to *ASCII files*. The *file number* identifies an *ASCII file* or, if zero, your terminal. When a zero file number is used, this statement is exactly equivalent to the PRINT USING statement.

The *format part* is either a *format string* represented as a *literal string* or *string variable* or it is a *statement number* referencing an IMAGE statement. Refer to the PRINT USING statement description for the form of a *format string* and to the IMAGE statement description for a *format string* in that statement.

The optional *using list* specifies the items to be printed: *string variables* or *numeric expressions* or *print functions* (LIN,TAB,SPA, or CTL) separated by commas. The list may be omitted only when the *format part* does not contain any D or A specifications.

Note that terminal dependent characters sent to an ASCII file produce effects different from those produced on a terminal. For example, all characters sent to an ASCII file that are longer than the line (record) width of the device are lost on an ASCII file. Also, the overprinting capability produced by a plus (+) as the carriage control character or by the functions LIN(0) or CTL(13) only work on terminals and certain line printers; all other ASCII file devices start a new line when requested to overprint.

These distinctions apply when printing to any ASCII file.

Examples:

```

110 PRINT #2;USING "S4D.DD";B
120 PRINT #2;USING 200;A$,B$(2,4)
130 PRINT #1;USING "PRINT EXAMPLE FOLLOWS"
135 C$="7D.2DE"
140 PRINT #1;USING C$;A,B,C
200 IMAGE 3X4A,2XA

```

PURGE Statement

General Form:

PURGE *return variable* , *file designator*

The PURGE statement deletes BASIC formatted or ASCII files from the system.

The *return variable* returns the status of the purge operation. The values returned and their meaning are as follows:

Return Value	Meaning
0	File successfully purged
1	File is busy and cannot be purged
2	File is not accessible
3	No such file

The *file designator* is a source string whose value is a *file name*.

An open file in your program cannot be purged until it has been closed. A file can be closed with an ASSIGN* statement.

The PURGE statement can be used to dissociate nonsharable devices from the name used in a FILE command. If the file is a disc file, the space it occupied is returned to the system.

A locked or private program saved in a group account library having the FCP (File Create/Purge) capability can be executed or chained-to in order to purge locked BASIC formatted files in group member accounts having the PFA (Program/File Access) capability. These files may be specified by appending ".idcode".

Examples:

```
10 PURGE N, "MYFILE"
20 PURGE N, "HERFIL.G707"
30 PURGE N, A$
```


READ Statement

General Form:

READ *read variable list*

The READ statement reads string and numeric values from DATA statements.

A *read variable list* is one or more *read variables* separated by commas. A *read variable* can be a *destination string* or a *numeric variable*.

Reading begins with the first item in the first DATA statement in the program. Subsequent reads continue with the first item not read. The type of *read variable* must agree with the data type in the DATA statement. You can use the TYP function to determine the next data type in a DATA statement.

Examples:

```
10 READ A,C(33), F1$(20,40)
20 READ G(I,J/2), Z$
```

READ # Statement



General Form:

```
READ # file number[ , record number][ ; read variable list]
```

The READ # statement reads data from BASIC formatted or ASCII files. The *file number* specifies a BASIC formatted or ASCII file, or, if the number is zero, your terminal. When *file number* is zero, READ # reads data from the terminal exactly as if it were an INPUT statement.

A serial file read statement (no *record number* specified) reads items from a file specified by *file number* into variables specified in the *read variable list*. A *read variable list* is one or more *read variables* separated by commas. A *read variable* is a *numeric variable* or a *destination string*. The first item read is the item following the current position of the file pointer. Record boundaries are ignored and the items read can begin in the middle of one record and end in the middle of another. If the *read variable list* is omitted, no action is taken.

A direct file read statement (*record number* specified) reads items from a specific record in a file into variables of the *read variable list*. If a record number is specified that is outside the range for the named file, an end-of-file condition occurs. If the *read variable list* is omitted, a direct file read moves the pointer to the beginning of the specified record, but does not read data. An attempt to read beyond the end of data in a record results in an end-of-record and an end-of-file condition. This causes a transfer to the IF END statement label if an IF END statement has been used.

The destination for string data must be a *destination string* and the destination for numeric data must be a *numeric variable*. If the data does not match the type of the destination variable, an error will occur. It is possible to check the type of the next data item with the TYP function described elsewhere in this section.

In both direct and serial file access, an attempt to read beyond a logical or physical end-of-file will cause the end-of-file condition to occur. Unless an IF END statement is used to transfer control to another statement in the program, an error will occur terminating the program.

READ # Statement (Cont)

Serial File Read Examples:

```

10 FILES AFILE,BFILE
20 DIM AS[5],XS[10],YS[10],CS[15]
30 DIM B[10]
40 MAT READ B
50 DATA 100,200,300,400,500,600,700,700,900,1000
60 LET AS="ABCDE"
70 PRINT #1;"ARRAY B"
74 FOR J=1 TO 10
75 PRINT #1;B[J]
76 NEXT J
80 PRINT #2;AS,B[3],B[6]
90 PRINT #2; END
100 PRINT #1;"END OF ARRAY"
110 READ #1,1
120 READ #2,1
130 READ #1;XS,A1,B1,C1,D1,E1
140 PRINT XS,LIN(1),A1,B1,C1,D1,E1
150 READ #1;A2,B2,C2,D2,E2
160 PRINT A2,B2,C2,D2,E2
170 READ #2;YS,A,B
180 PRINT YS,A,B
190 READ #1;CS
195 PRINT CS
200 READ #1;X
210 REM..ATTEMPT TO READ END-OF-FILE CAUSES TERMINATION
220 END
RUN

```

```

ARRAY B
  100          200          300          400          500
  600          700          700          900          1000
ABCDE         300          600
END OF ARRAY

```

END-OF-FILE/END OF RECORD IN LINE 200

READ # Statement (Cont)

After data is written on files 1 and 2 with the print statements in lines 70 through 100, and the pointer is restored to the start of each file in lines 110 and 120, the data is ready to be read. The first six items in file 1 are read in line 130. The next five items are read in line 150. The three items written on file 2 are read in line 170. The PRINT statements are inserted to demonstrate the accuracy of the reads and the previous writes. A string item remains in file 1 and is read in line 190. Line 200 attempts to read an end-of-file mark causing the message: END OF FILE/END OF RECORD IN LINE 200 to be printed.

Direct File Read Examples:

```

5  FILES AFILE
10  DIM C(5),D(5)
20  DIM AS(20),XS(20)
25  MAT C=CON
30  READ A,B,C
40  DATA 1,2,3
50  PRINT #1,2;A,B,C,"NUMBERS"
60  PRINT #1,1;"ARRAY C"
70  FOR I=1 TO 5
80  PRINT #1;C[I]
90  NEXT I
100 READ #1,1;XS
110 FOR I=1 TO 5
120 READ #1;D[I]
130 NEXT I
140 PRINT XS
150 MAT PRINT D
160 READ #1,2;D,E,F,AS
170 PRINT AS,D;E;F
180 END
RUN

```

```

ARRAY C
  1
  1
  1
  1
  1

NUMBERS          1      2      3

DONE

```

ASCII File Read Operations

Reading from an ASCII file follows the same rules as the INPUT statement. The data is read as if it were being input at your terminal.

REC Function

General Form:

REC (numeric expression)

The REC function returns the current record number being accessed in the file specified.

The *numeric expression* is evaluated and rounded to an integer. This integer is used as the *file number*. The REC function cannot be used with ASCII files. If attempted the program will be terminated with an error. Specification of an invalid *file number* will also result in an error.

Examples:

```
20 PRINT "I AM NOW AT RECORD " REC(N) "IN FILE " N
30 R=REC(N+1)
```

REM Statement

General Form:

REM [remark]

The REM statement allows you to add comments to your program.

The optional *remark* may be any series of characters. The REM statements are not executed.

Examples:

```
200 REM****ITEM SEARCH SUBROUTINE****
300 REM This program is used to sort data by value
400 REM THIS PORTION OF THE PROGRAM IS ONLY
401 REM EXECUTED IF THE DATA IS INVALID
402 REM AND THE ERROR DETECT FLAG IS SET
500 REMARK — "ARK" are the first three characters of this REM statement
```

RESTORE Statement

General Form:

RESTORE [*statement number*]

The RESTORE statement resets the program's data pointer to the first item in a DATA statement.

The optional *statement number* indicates a specific DATA statement. If no *statement number* is given, the pointer is reset to the first data item in the first DATA statement in the program.

If the *statement number* does not reference a DATA statement, the next READ or MAT READ statement will cause the pointer to be moved to the first DATA statement following the referenced statement. If there is no following DATA statement, the attempted read operation will terminate the program with an error.

Examples:

```
20 RESTORE 30
```

```
30 RESTORE
```

RND Function

General Form:

RND (*numeric expression*)

The RND function is a numeric valued function which returns a pseudo random number in the range $0 \leq R < 1$.

The number is not truly random but is calculated from an initial or "seed" number. Each succeeding number in the sequence is then calculated from the previous number. An initial seed number (based on the date and time of day) is assigned to each port on the system every time the system is started. The sequence may be altered by specifying a new seed. This allows you to repeat given sequences of random numbers.

The *numeric expression* is evaluated and used to obtain the next random number based on the last number or to change the last number and begin a new sequence as follows:

Value of the <i>Numeric Expression</i>	Result
less than 0	A pseudo random number is returned that is calculated from the <i>numeric expression</i> .
greater than or equal to 0	A pseudo random number is returned that is calculated from the last number generated.

In order to generate a repeatable sequence of pseudo random numbers, you should first use the RND function with a *numeric expression* that has a value < 0 , and then continue using the function with a *numeric expression* that has a value ≥ 0 . To repeat the sequence, use the value from the initial call.

Examples:

```
100 N3=RND(-5.55)
110 D =RND(F)** 10
```

SGN Function

General Form:

SGN (*numeric expression*)

The SGN function is a numeric valued function which returns a number indicating the sign of the *numeric expression*. If the sign of the *numeric expression* (after evaluation) is positive, the value returned is 1. If the sign is negative, the value returned is -1. If the numeric expression evaluates to zero, the value returned is 0. For instance, $\text{SGN}(-3.14) = -1$, $\text{SGN}(943.2) = 1$, and $\text{SGN}(0) = 0$.

Examples:

```
1200 A4 = SGN (P2)
```

```
1210 A(I) = SGN (A2+A(I-1))
```

SIN Function

General Form:

SIN (*numeric expression*)

The SIN function is a numeric valued function which returns the sine of the *numeric expression*. The *numeric expression* is interpreted as being in radians. If the absolute value of the *numeric expression* exceeds approximately 102900, your program will be terminated with an error.

Example:

```
440 LET S(I) = SIN(3.1415/I)
```


SPA Function

General Form:

SPA (numeric expression)

The SPA function is used in a PRINT, PRINT# PRINT USING, PRINT# USING, MAT PRINT, MAT PRINT#, MAT PRINT USING, or MAT PRINT# USING statement to print the number of blank characters specified by the *numeric expression*.

The *numeric expression* is evaluated and rounded to an integer. The SPA function is ignored when the *numeric expression* is zero or negative. If the number of blanks will not fit on the current line, or if *numeric expression* evaluates to a value greater than 71, an X-OFF, carriage return, and line feed are generated. Formatted print statements using the SPA function do not have a line length limit of 71 characters. (Line printers and other ASCII devices can have more than 72 characters.) SPA is ignored for file prints (PRINT#, MAT PRINT#) to BASIC formatted files, but is allowed for ASCII files. Note that a comma after a SPA function is treated like a semicolon (see *print delimiter* under PRINT). To determine how many spaces are printed by the system when a semicolon delimiter is used, refer to Numeric Output formats in the PRINT statement description.

Example:

```
10 PRINT A$, SPA(10), B$, SPA(X+LEN(C$)), D$
```

SQR Function

General Form:

SQR (numeric expression)

The SQR function is a numeric valued function which returns the square root of the *numeric expression*. The *numeric expression* must evaluate to a number greater than or equal to 0; otherwise, your program will terminate with an error.

Examples:

```
932 LET S2 = SQR(2)
940 PRINT SQR (B**2 + C **2)
```

STOP Statement

General Form:

STOP

The STOP statement terminates execution of the program and returns control to the system. The STOP statement may occur at any point in the program except the last statement which must be END.

Normal termination will not occur if a device error prevents completion of an output operation to a device such as the line printer or punch. To allow termination, correct the device error. The BREAK key is disabled in this situation.

Example:

```
300 STOP
```

SYS Function

General Form:

SYS(numeric expression)

The SYS function is a numeric valued expression that returns a value depending on the *numeric expression*. The SYS function is used either with the IF ERROR statement for programmatic detection of errors, or for programmatic detection of the BREAK key after it has been disabled by the BRK function. It also returns the terminal type of the terminal where the program is being executed.

The value of the numeric expression is 0, 1, 2, 3, or 4 and determines the operation of SYS as shown below.

- SYS(0) Value returned is either an error number, or zero indicating no error has occurred. Following execution, value is reset to zero.
- SYS(1) Value returned is either the line number in which last error occurred, or zero if no error occurred; following execution, value is reset to zero.
- SYS(2) Value returned is the file number of the last file accessed (1-16) or zero if the last file accessed was a user terminal; if no file or terminal was accessed, minus one (-1) is returned.
- SYS(3) Value returned is either a one indicating that the BREAK key was pressed since it was disabled or since SYS(3) was last executed, or a zero indicating that the BREAK key has not been pressed in that interval; following execution, value is reset to zero. SYS(3) is only operative when the normal use of the BREAK key has been disabled by the BRK function (refer to BRK in this section).
- SYS(4) Value returned is 0 through 9 indicating the terminal type at log in (refer Appendix D for details of different terminal types).

When a CHAIN statement is executed transferring control to another program, all SYS function values are reset to zero.

Refer to Appendix C for a list of the BASIC errors and the error number associated with each error. Note that the errors are separated into four categories: programming, format, warning, and file, and that the error numbers for these categories are assigned in increments of 100.

Examples

```
1010 PRINT "FILE ERROR"SYS(0)"IN LINE"SYS(1)"OF FILE"SYS(2)
1050 IF SYS(3) <> 0 THEN 1090
```

SYSTEM Statement

General Form:

SYSTEM return variable , source string

or

SYSTEM destination string , source string

The SYSTEM statement allows you to execute some operating system commands from a running program.

The command is given in the *source string* and must be in the same form that would be used to enter the command normally. The *OUT=file name* construct allowed in some commands is not allowed in the SYSTEM statement. Any lower case letters used in the *source string* are shifted to upper case.

Commands which can be used with the first form of the SYSTEM statement are BYE, ECHO, MESSAGE, FILE, PROTECT, LOCK, PRIVATE, UNRESTRICT, MWA, SWA and PAUSE. The *return variable* is set to 0 if the command is executed and 1 if it is not.

A locked or private program saved in a group account library having the FCP (File Create/Purge) capability can be executed or chained to in order to specify MWA (multiple write access) for a locked BASIC formatted file using the SYSTEM statement. The file must belong to a group member having the PFA (Program/File Access) capability. The group account library must have the MWA (Multiple Write Access) capability.

Example:

```
SYSTEM R, "MWA - GRTH.C402"
```

Commands which can be used with the second form of the SYSTEM statement are TIME, CATALOG, GROUP, LIBRARY, and LENGTH. These commands would normally produce output on your terminal. Only the first line of this output (less any heading) will be returned in the *destination string*. Note that for the CATALOG, GROUP, and LIBRARY commands you can enter start print parameter (refer to the descriptions for these commands in Section 10).

In the example given in line 30 below, the first line of actual data, starting with the first entry equal to or greater than "T" will be returned as the value of A\$.

Examples:

```
10 SYSTEM R0, "FILE-LPR,LP0"  
20 SYSTEM A$, B$  
30 SYSTEM A$, "CAT-T"  
40 SYSTEM P0$(1,4),"TIM"
```

TAB Function

General Form:

TAB(*numeric expression*)

The TAB function is used in print operations to move the current print position to a specified column.

The TAB function can be used in PRINT, PRINT=, PRINT USING, PRINT= USING, MAT PRINT, MAT PRINT =, MAT PRINT USING and MAT PRINT = USING statements. The *numeric expression* is evaluated and rounded to an integer to obtain the destination column. Print columns are numbered 0 to 71 (note that line printers and other ASCII devices can have more than 72 columns). If the *numeric expression* evaluates to a column position lower than the current print position, the TAB function is ignored. If the *numeric expression* evaluates to more than 71, the print position is moved to the beginning of the next line (RETURN) unless this occurs in a PRINT USING or MAT PRINT USING statement. In formatted print statements, X-OFF, carriage return, line feed are not generated if the TAB argument exceeds 71.

The TAB function is ignored for file prints (PRINT = and MAT PRINT =) to BASIC formatted files but is allowed for ASCII files.

Commas following TAB functions in print statements are treated like semicolons (refer to the discussion of *print delimiter* under the PRINT STATEMENT). To determine the last print position used prior to use of TAB, refer to Numeric Output formats in the PRINT statement description.

Examples:

```
10 PRINT A,B$(1,10),TAB (40)
20 PRINT TAB(A+B),N,TAB(C),M
```

TAN Function

General Form:

TAN (*numeric expression*)

The TAN function is a numeric valued function that returns the tangent of the *numeric expression*.

The *numeric expression* is interpreted as being in radians. If the absolute value of the *numeric expression* exceeds approximately 51500, your program will be terminated with an error.

Example:

```
14 A3 = TAN(3.1415/P2)
```

TIM Function

General Form:

TIM (numeric expression)

The TIM function is a numeric valued function which returns the current second, minute, hour, day, or year.

The *numeric expression* is evaluated and rounded to an integer. This integer is used to specify which of the time values is to be returned as follows:

Value of Expression	Time Returned
0	current minute (0-59)
1	current hour (0-23)
2	current day (1-366)
3	current year (0-99)
4	current second (0-59)

Note the difference between the TIM function and the TIME command. The TIME command is used outside a program and gives the current console time used for that port, the total time used to date for that account, and the total time permitted for that account. The TIM function must be used within a program and returns the current time based on the system clock.

Examples:

```
20 IF TIM(0)-A = 15 THEN 9000
30 A3= TIM(B)
40 PRINT "SECOND";TIM(4);"MINUTE";TIM(0);"HOUR";TIM(1);"DAY";TIM(2)
50 PRINT TIM(2*X/A-2)
```

TRN Function

(Refer to MAT. . . TRN)

TYP Function

General Form:

TYP (*numeric expression*)

The TYP function is a numeric valued function that returns the data type of the next sequential item in a file or data statement.

The *numeric expression* is evaluated and rounded to an integer. The absolute value of the integer must be a valid *file number* or 0. If it is not, the program will be terminated with an error.

When the TYP function is used to test data in the file indicated by the *numeric expression* it will return a value indicating that the next item in the file is a number, string, end-of-file mark, or end-of-record mark. If a positive argument is used in the TYP function and the referenced file contains an end-of-record mark as the next item, the next record is examined. This allows the TYP function to be used with serial files.

When used to test data contained in a program DATA statement, the *numeric expression* must have a value of 0.

Return values for various values of the *numeric expression* are as follows:

<u>Return Value</u>	<u>Expression < 0 (file test)</u>	<u>Expression > 0 (file test)</u>	<u>Expression = 0 (DATA statement test)</u>
1	number	number	number
2	string	string	string
3	end of file	end of file	end of data
4	end of record	—	—

Note that ASCII files will return values of 2 only.

Examples:

```
10 A = TYP(-1)
20 GOTO TYP(-X) OF 40,50,6070
30 PRINT TYP(X*2)
```

UNLOCK Statement

General Form:

UNLOCK # file number [, return variable]

The UNLOCK statement clears a file status flag that has been previously set with a LOCK statement (refer to the LOCK statement).

The UNLOCK statement should be used to release a file as soon as the file access operation is complete to allow others using the locking technique to access the file. If the optional *return variable* is used you can check the values returned to determine the result of the unlocking operation. A list of return values and their meaning is given below.

Return Value	Meaning
0	File successfully unlocked
1	File already unlocked (either by this or another program)
2	File number invalid

The UNLOCK statement does not affect LOCK statements in other programs. The successful execution of an UNLOCK statement leaves the file pointer positioned at the end of the last accessed record in the file.

Use of the LOCK and UNLOCK statements guarantees that any data you write to the file while the file is locked is written to the disc before the file is unlocked.

Examples:

```
10 UNLOCK #3,Z0  
20 UNLOCK # N
```

UPDATE Statement

General Form:

UPDATE # *file number* ; *numeric expression*

or

UPDATE # *file number* ; *source string*

The UPDATE statement replaces the next sequential data item in the file referenced by the *file number* with the value of either the *numeric expression* or the *source string*.

The data type of the new item must match the type (numeric or string) of the item being replaced. The *file number* must refer to a BASIC formatted file. The UPDATE statement cannot be used with ASCII files. The *source string* must be the same size as the file string being replaced. If the *source string* is shorter than the old string, it will be padded on the right with blanks. If the *source string* is longer than the old string it will be truncated from the right to fit.

If the next item on the file is an end-of-record mark, the next record is read from the disc before the update takes place. If the next item is an end-of-file mark, an end-of-file condition occurs if an update is attempted. You must have write access to the file to use the UPDATE statement.

Examples:

```
10 UPDATE #3 ; A$
20 UPDATE #1 ; "JIM"
30 UPDATE #7 ; Z1$(3,20)
40 UPDATE #2 ; A(N,M)
50 UPDATE #4 ; X*12/52
```


UPS\$ Function

General Form:

UPS\$ (source string)

The UPS\$ function returns a string equivalent of the *source string* with all characters shifted to upper case.

Only the 26 lower case alphabetic characters (a to z) are effected. For example, UPS\$("abc%12") would return the string "ABC%12".

Examples:

```
200 PRINT UPS$(G$)
300 Q$=UPS$(R$(j,k))
```

ZER Function

(Refer to MAT. .ZER)

USING THE HP 2000 CHARACTER SET

APPENDIX

A

Most terminals used with the HP 2000 System represent characters in ASCII (American Standard Code for Information Interchange) code. Also, the system uses this code to store all string data. The ASCII character set is given in table A-1. An alternate code used by the IBM 2741 Communication terminal is discussed in Appendix D.

The 128 character set in table A-1 is ranked according to the decimal equivalent of the code. This decimal value is used in determining the relative value of strings.

Note that in table A-1, characters with values 0 through 32 do not have a graphic or printing character. They are typically used to perform a variety of terminal and communication operations. Characters with values between 0 and 63 can be generated by pressing several keys on your terminal simultaneously. This technique is used to produce special ASCII characters that may not appear as separate keys on your terminal. This is important since the character set available on terminal keyboards may vary, for instance some terminals have no *line feed* key. A line feed may be entered, however, by holding down the control key and pressing the J key (J^c).

The full 2000 character set consists of 256 characters. The first 128 characters are ASCII code; EBCDIC code requires all 256 characters. The full character set is shown in Table A-2. Both ASCII and EBCDIC are shown where applicable, together with the numeric equivalents of the codes in decimal, octal, and hexadecimal. The card codes for the 029 punch for both ASCII and EBCDIC are also provided.

Table A-2 is useful when using the RJE facility to transmit data to or from the IBM host system. In general, when a character is sent from a 2000 system to an IBM host, each character is translated to its EBCDIC equivalent. That is, a letter A sent from 2000 is received as an A at the host site. However, an A in ASCII is a decimal 65 whereas an A in EBCDIC is a decimal 193. Therefore, the 2000 system does a translation and sends the A as a decimal 193. Reception is the obverse; that is, the IBM host sends a decimal 193 when it wants to send the letter A and the 2000 system translates it to a decimal 65 so the user receives an ASCII A. To determine the numeric equivalent of a character transmitted between these two systems, find the graphic representation of the ASCII character for HP 2000 and the EBCDIC graphic for IBM in Table A-2. The associated numeric value is used in the transmission. All exceptions are noted in Table A-3.

CDC uses a 63 or 64 character set that generally corresponds to the standard HP 2000 printable characters, decimal code 32 (space) through decimal code 95 (underscore). With the exceptions listed in Table A-3, the CDC and HP 2000 graphics correspond. Wherever a 2000 graphic is not represented in the CDC character set, it is converted to a blank when sent to CDC. Since only 6-bit characters can be sent from a CDC system, every CDC character may be translated to ASCII with the exception of those characters listed in Table A-3.

When reading cards using a BASIC program, refer to Table A-2 for a list of valid character codes. Note that the lower case alphabetic characters cannot be read by a BASIC program.

Table A-1. ASCII Character Set

Decimal Value	Control/Graphic	Comments	Decimal Value	Control/Graphic	Comments
0	NUL @ ^C	Null	64	@	Commercial at
1	SOH A ^C	Start of heading	65	A	Uppercase A
2	STX B ^C	Start of text	66	B	Uppercase B
3	ETX C ^C	End of text	67	C	Uppercase C
4	EOT D ^C	End of transmission	68	D	Uppercase D
5	ENQ E ^C	Enquiry	69	E	Uppercase E
6	ACK F ^C	Acknowledge	70	F	Uppercase F
7	BEL G ^C	Bell	71	G	Uppercase G
8	BS H ^C	Backspace	72	H	Uppercase H
9	HT I ^C	Horizontal tabulation	73	I	Uppercase I
10	LF J ^C	Line feed	74	J	Uppercase J
11	VT K ^C	Vertical tabulation	75	K	Uppercase K
12	FF L ^C	Form feed	76	L	Uppercase L
13	CR M ^C	Carriage return	77	M	Uppercase M
14	SO N ^C	Shift out	78	N	Uppercase N
15	SI O ^C	Shift in	79	O	Uppercase O
16	DLE P ^C	Data link escape	80	P	Uppercase P
17	DC1 Q ^C	Device control 1 (X-ON)	81	Q	Uppercase Q
18	DC2 R ^C	Device control 2	82	R	Uppercase R
19	DC3 S ^C	Device control 3 (X-OFF)	83	S	Uppercase S
20	DC4 T ^C	Device control 4	84	T	Uppercase T
21	NAK U ^C	Negative acknowledge	85	U	Uppercase U
22	SYN V ^C	Synchronous idle	86	V	Uppercase V
23	ETB W ^C	End of transmission block	87	W	Uppercase W
24	CAN X ^C	Cancel	88	X	Uppercase X
25	EM Y ^C	End of medium	89	Y	Uppercase Y
26	SUB Z ^C	Substitute	90	Z	Uppercase Z
27	ESC [^C	Escape	91	{	Opening bracket
28	FS \ ^C	File separator	92	\	Reverse slant
29	GS] ^C	Group separator	93	}	Closing bracket
30	RS ^ ^C	Record separator	94	^	Circumflex
31	US _ ^C	Unit separator	95	_	Underscore
32	SP ^V	Space (blank)	96	`	Grave accent
33	! a ^C	Exclamation point	97	a	Lowercase a
34	" b ^C	Quotation mark	98	b	Lowercase b
35	# c ^C	Number sign	99	c	Lowercase c
36	\$ d ^C	Dollar sign	100	d	Lowercase d
37	% e ^C	Percent sign	101	e	Lowercase e
38	& f ^C	Ampersand	102	f	Lowercase f
39	' g ^C	Apostrophe	103	g	Lowercase g
40	(h ^C	Left parenthesis	104	h	Lowercase h
41) i ^C	Right parenthesis	105	i	Lowercase i
42	* j ^C	Asterisk	106	j	Lowercase j
43	+ k ^C	Plus sign	107	k	Lowercase k
44	, l ^C	Comma	108	l	Lowercase l
45	- m ^C	Minus sign (hyphen)	109	m	Lowercase m
46	. n ^C	Decimal point (period)	110	n	Lowercase n
47	/ o ^C	Slash	111	o	Lowercase o
48	0 p ^C	Zero	112	p	Lowercase p
49	1 q ^C	One	113	q	Lowercase q
50	2 r ^C	Two	114	r	Lowercase r
51	3 s ^C	Three	115	s	Lowercase s
52	4 t ^C	Four	116	t	Lowercase t
53	5 u ^C	Five	117	u	Lowercase u
54	6 v ^C	Six	118	v	Lowercase v
55	7 w ^C	Seven	119	w	Lowercase w
56	8 x ^C	Eight	120	x	Lowercase x
57	9 y ^C	Nine	121	y	Lowercase y
58	: z ^C	Colon	122	z	Lowercase z
59	; { ^C	Semicolon	123	{	Left brace
60	< ^C	Less than	124		Vertical line
61	= } ^C	Equals sign	125	}	Right brace
62	> ~	Greater than	126	~	Tilde
63	? DEL ^C	Question mark	127	DEL	Delete

Table A-2. 2000 ASCII/EBCDIC Character Set

Values			Control/Graphic		Card Codes	
Dec	Oct	Hex	ASCII/CTL	EBCDIC	029 ASCII	029 EBCDIC
0	0	00	NUL @ ^c	NUL		12-0-1-8-9
1	1	01	SOH A ^c	SOH		12-1-9
2	2	02	STX B ^c	STX		12-2-9
3	3	03	ETX C ^c	ETX		12-3-9
4	4	04	EOT D ^c	PF		12-4-9
5	5	05	ENQ E ^c	HT		12-5-9
6	6	06	ACK F ^c	LC		12-6-9
7	7	07	BEL G ^c	DEL		12-7-9
8	10	08	BS H ^c			12-8-9
9	11	09	HT I ^c			12-1-8-9
10	12	0A	LF J ^c	SMM		12-2-8-9
11	13	0B	VT K ^c	VT		12-3-8-9
12	14	0C	FF L ^c	FF		12-4-8-9
13	15	0D	CR M ^c	CR		12-5-8-9
14	16	0E	SO N ^c	SO		12-6-8-9
15	17	0F	SI O ^c	SI		12-7-8-9
16	20	10	DLE P ^c	DLE		12-11-8-9
17	21	11	DC1 Q ^c	DC1		11-1-9
18	22	12	DC2 R ^c	DC2		11-2-9
19	23	13	DC3 S ^c	TM		11-3-9
20	24	14	DC4 T ^c	RES		11-4-9
21	25	15	NAK U ^c	NL		11-5-9
22	26	16	SYN V ^c	BS		11-6-9
23	27	17	ETB W ^c	IL		11-7-9
24	30	18	CAN X ^c	CAN		11-8-9
25	31	19	EM Y ^c	EM		11-1-8-9
26	32	1A	SUB Z ^c	CC		11-2-8-9
27	33	1B	ESC [^c	CU1		11-3-8-9
28	34	1C	FS \ ^c	IFS		11-4-8-9
29	35	1D	CS] ^c	IGS		11-5-8-9
30	36	1E	RS ^ ^c	IRS		11-6-8-9
31	37	1F	US _ ^c	IUS		11-7-8-9
32	40	20	space	DS	none	11-0-1-8-9
33	41	21	"	SOS	11-2-8	0-1-9
34	42	22	"	FS	7-8	0-2-9
35	43	23	"		3-8	0-3-9
36	44	24	\$	BYP	11-3-8	0-4-9
37	45	25	%	LF	0-4-8	0-5-9
38	46	26	&	ETB	12	0-6-9
39	47	27	'	ESC	5-8	0-7-9
40	50	28	(12-5-8	0-8-9
41	51	29)		11-5-8	0-1-8-9
42	52	2A	*	SM	11-4-8	0-2-8-9
43	53	2B	+	CU2	12-6-8	0-3-8-9
44	54	2C	,		0-3-8	0-4-8-9
45	55	2D	;	ENQ	11	0-5-8-9
46	56	2E	:	ACK	12-3-8	0-6-8-9
47	57	2F	?	BEL	0-1	0-7-8-9
48	60	30	0		0	12-11-0-1-8-9
49	61	31	1		1	1-9
50	62	32	2	SYN	2	2-9
51	63	33	3		3	3-9
52	64	34	4	PN	4	4-9
53	65	35	5	RS	5	5-9
54	66	36	6	UC	6	6-9
55	67	37	7	EOT	7	7-9
56	70	38	8		8	8-9
57	71	39	9		9	1-8-9
58	72	3A	:		2-8	2-8-9
59	73	3B	;	CU3	11-6-8	3-8-9
60	74	3C	<	DC4	12-4-8	4-8-9
61	75	3D	=	NAK	6-8	5-8-9
62	76	3E	>		0-6-8	6-8-9
63	77	3F	?	SUB	0-7-8	7-8-9

Values			Control/Graphic		Card Codes	
Dec	Oct	Hex	ASCII	EBCDIC	029 ASCII	029 EBCDIC
64	100	40	@	space	4-8	no punch
65	101	41	A		12-1	12-0-1-9
66	102	42	B		12-2	12-0-2-9
67	103	43	C		12-3	12-0-3-9
68	104	44	D		12-4	12-0-4-9
69	105	45	E		12-5	12-0-5-9
70	106	46	F		12-6	12-0-6-9
71	107	47	G		12-7	12-0-7-9
72	108	48	H		12-8	12-0-8-9
73	111	49	I		12-9	12-1-8
74	112	4A	J	c	11-1	12-2-8
75	113	4B	K	c	11-2	12-3-8
76	114	4C	L	<	11-3	12-4-8
77	115	4D	M	(11-4	12-5-8
78	116	4E	N	+	11-5	12-6-8
79	117	4F	O	,	11-6	12-7-8
80	120	50	P	&	11-7	12
81	121	51	Q		11-9	12-11-9
82	122	52	R		11-9	12-11-2-9
83	123	53	S		0-2	12-11-3-9
84	124	54	T		0-3	12-11-4-9
85	125	55	U		0-4	12-11-5-9
86	126	56	V		0-5	12-11-6-9
87	127	57	W		0-6	12-11-7-9
88	130	58	X		0-7	12-11-8-9
89	131	59	Y		0-8	11-1-8
90	132	5A	Z	!	0-9	11-2-8
91	133	5B	[!"#\$	12-2-8	11-3-8
92	134	5C	\	!"#\$	0-2-8	11-4-8
93	135	5D]	!"#\$	11-2-8	11-5-8
94	136	5E	^	!"#\$	11-7-8	11-6-8
95	137	5F	_	!"#\$	0-5-8	11-7-8
96	140	60	`	!"#\$		11
97	141	61	a	!"#\$		0-1
98	142	62	b	!"#\$		11-0-2-9
99	143	63	c	!"#\$		11-0-3-9
100	144	64	d	!"#\$		11-0-4-9
101	145	65	e	!"#\$		11-0-5-9
102	146	66	f	!"#\$		11-0-6-9
103	147	67	g	!"#\$		11-0-7-9
104	150	68	h	!"#\$		11-0-8-9
105	151	69	i	!"#\$		0-1-8
106	152	6A	j	!"#\$		12-11
107	153	6B	k	!"#\$		0-3-8
108	154	6C	l	!"#\$		0-4-8
109	155	6D	m	!"#\$		0-5-8
110	156	6E	n	!"#\$		0-6-8
111	157	6F	o	!"#\$		0-7-8
112	160	70	p	!"#\$		12-11-0
113	161	71	q	!"#\$		12-11-0-1-9
114	162	72	r	!"#\$		12-11-0-2-9
115	163	73	s	!"#\$		12-11-0-3-9
116	164	74	t	!"#\$		12-11-0-4-9
117	165	75	u	!"#\$		12-11-0-5-9
118	166	76	v	!"#\$		12-11-0-6-9
119	167	77	w	!"#\$		12-11-0-7-9
120	170	78	x	!"#\$		12-11-0-8-9
121	171	79	y	!"#\$		1-8
122	172	7A	z	!"#\$		2-8
123	173	7B	{	!"#\$		3-8
124	174	7C		!"#\$		4-8
125	175	7D	}	!"#\$		5-8
126	176	7E	~	!"#\$		6-8
127	177	7F	DEL	!"#\$		7-8

Table A-2. 2000 ASCII/EBCDIC Character Set (Cont)

Values			Control/Graphic		Card Codes		
Dec	Oct	Hex	ASCII	EBCDIC	029 ASCII	029 EBCDIC	
128	200	80				12 0 1 8	
129	201	81		a		12 0 1	
130	202	82		b		12 0 2	
131	203	83		c		12 0 3	
132	204	84		d		12 0 3	
133	205	85		e		12 0 4	
134	206	86		f		12 0 5	
135	207	87		g		12 0 6	
136	210	88		h		12 0 7	
137	211	89		i		12 0 8	
138	212	8A				12 0 2 8	
139	213	8B				12 0 3 8	
140	214	8C				12 0 4 8	
141	215	8D				12 0 5 8	
142	216	8E				12 0 6 8	
143	217	8F				12 0 7 8	
144	220	90				12 11 1 8	
145	221	91		j		12 11 1	
146	222	92		k		12 11 2	
147	223	93		l		12 11 3	
148	224	94		m		12 11 4	
149	225	95		n		12 11 5	
150	226	96		o		12 11 6	
151	227	97		p		12 11 7	
152	230	98		q		12 11 8	
153	231	99		r		12 11 9	
154	232	9A				12 11 2 8	
155	233	9B				12 11 3 8	
156	234	9C				12 11 4 8	
157	235	9D				12 11 5 8	
158	236	9E				12 11 6 8	
159	237	9F				12 11 7 8	
160	240	A0				11 0 1 8	
161	241	A1				11 0 1	
162	242	A2		s		11 0 2	
163	243	A3		t		11 0 3	
164	244	A4		u		11 0 4	
165	245	A5		v		11 0 5	
166	246	A6		w		11 0 6	
167	247	A7		x		11 0 7	
168	250	A8		y		11 0 8	
169	251	A9		z		11 0 9	
170	252	AA				11 0 2 8	
171	253	AB				11 0 3 8	
172	254	AC				11 0 4 8	
173	255	AD				11 0 5 8	
174	256	AE				11 0 6 8	
175	257	AF				11 0 7 8	
176	260	B0		0		12 11 0 1 8	
177	261	B1		1		12 11 0 1	
178	262	B2		2		12 11 0 2	
179	263	B3		3		12 11 0 3	
180	264	B4		4		12 11 0 4	
181	265	B5		5	} lower-case numerals	12 11 0 5	
182	266	B6		6		12 11 0 6	
183	267	B7		7		12 11 0 7	
184	270	B8		8		12 11 0 8	
185	271	B9		9		12 11 0 9	
186	272	BA					12 11 0 2 8
187	273	BB					12 11 0 3 8
188	274	BC					12 11 0 4 8
189	275	BD					12 11 0 5 8
190	276	BE				12 11 0 6 8	
191	277	BF				12 11 0 7 8	

Values			Control/Graphic		Card Codes	
Dec	Oct	Hex	ASCII	EBCDIC	029 ASCII	029 EBCDIC
192	300	C0				12 0
193	301	C1		A		12 1
194	302	C2		B		12 2
195	303	C3		C		12 3
196	304	C4		D		12 4
197	305	C5		E		12 5
198	306	C6		F		12 6
199	307	C7		G		12 7
200	310	C8		H		12 8
201	311	C9		I		12 9
202	312	CA				12 0 2 8 9
203	313	CB				12 0 3 8 9
204	314	CC				12 0 4 8 9
205	315	CD				12 0 5 8 9
206	316	CE				12 0 6 8 9
207	317	CF				12 0 7 8 9
208	320	D0				11 0
209	321	D1		J		11 1
210	322	D2		K		11 2
211	323	D3		L		11 3
212	324	D4		M		11 4
213	325	D5		N		11 5
214	326	D6		O		11 6
215	327	D7		P		11 7
216	330	D8		Q		11 8
217	331	D9		R		11 9
218	332	DA				12 11 2 8 9
219	333	DB				12 11 3 8 9
220	334	DC				12 11 4 8 9
221	335	DD				12 11 5 8 9
222	336	DE				12 11 6 8 9
223	337	DF				12 11 7 8 9
224	340	E0				0 2 8
225	341	E1				11 0 1 9
226	342	E2		S		0 2
227	343	E3		T		0 3
228	344	E4		U		0 4
229	345	E5		V		0 5
230	346	E6		W		0 6
231	347	E7		X		0 7
232	350	E8		Y		0 8
233	351	E9		Z		0 9
234	352	EA				11 0 2 8 9
235	353	EB				11 0 3 8 9
236	354	EC				11 0 4 8 9
237	355	ED				11 0 5 8 9
238	356	EE				11 0 6 8 9
239	357	EF				11 0 7 8 9
240	360	F0		0		0
241	361	F1		1		1
242	362	F2		2		2
243	363	F3		3		3
244	364	F4		4		4
245	365	F5		5		5
246	366	F6		6		6
247	367	F7		7		7
248	370	F8		8		8
249	371	F9		9		9
250	372	FA				12 11 0 2 8 9
251	373	FB				12 11 0 3 8 9
252	374	FC				12 11 0 4 8 9
253	375	FD				12 11 0 5 8 9
254	376	FE				12 11 0 6 8 9
255	377	FF				12 11 0 7 8 9

Table A-3. Differences Between HP 2000, IBM, and CDC Graphics



TRANSMITTED/ RECEIVED BY HP 2000		RECEIVED/ TRANSMITTED BY IBM		COMMENTS
VALUES DEC OCT HEX	HP 2000 GRAPHIC	IBM GRAPHIC	VALUES DEC OCT HEX	
33 41 21	!		79 117 4F	HP 2000 exclamation point – IBM vertical bar
91 133 5B	[¢	74 112 4A	HP 2000 left bracket – IBM cent
92 134 5C	\	(none)	224 340 E0	HP 2000 back slash – no IBM graphic exists
93 135 5D]	!	90 132 5A	HP 2000 right bracket – IBM exclamation mark
94 136 5E	^	¬	95 137 5F	HP 2000 circumflex – IBM not
TRANSMITTED/ RECEIVED BY HP 2000		RECEIVED/ TRANSMITTED BY CDC		COMMENTS
VALUES DEC OCT HEX	HP 2000 GRAPHIC	CDC GRAPHIC	VALUES	
33 41 21	!	^		HP 2000 exclamation point – CDC caret
34 42 22	"	≠		HP 2000 quotation mark – CDC not equals sign
35 43 23	#	≡		HP 2000 pound sign – CDC identity
37 45 25	%	: ¹		HP 2000 percent – CDC colon (63-character set)
38 46 26	&	^		HP 2000 ampersand – CDC circumflex
39 47 27	'	↑		HP 2000 apostrophe – CDC up arrow
63 77 3F	?	↓		HP 2000 question mark – CDC down arrow
64 100 40	@	≤		HP 2000 commercial at – CDC less than or equals
92 134 5C	\	≥		HP 2000 back slash – CDC greater than or equals
94 136 5E	_	→		HP 2000 underline – CDC right arrow
1. HP 2000 % is transmitted or received as a CDC colon when the CDC 63-character set is used; with the CDC 64-character set, a % is received or transmitted.				
The CDC numeric values depend on the host system and can vary. Therefore, they are not listed.				

HOW TO PREPARE A PAPER TAPE OFF-LINE

APPENDIX

B

To prepare a BASIC program on paper tape for input:

1. Set terminal status to "LOCAL."
2. Press the ON button on the paper tape punch.
3. Press control-@ (or the HERE IS key if available) several times to put leading feed holes on the tape.
4. Type the program as usual, *terminating each line with X-OFF (S^c), return, line feed.*
5. Press control-@ (or HERE IS) several times to put trailing feed holes on the tape.
6. Press the OFF button on the paper tape punch.

The standard on-line editing features, such as line delete and character delete, may be punched on paper tape.

Pressing the BACKSPACE button on the paper tape punch, then the RUBOUT or DEL key on the keyboard, physically deletes the previous character from the paper tape.

An alternate method of producing a punched paper tape copy of a program is to use the PUNCH command. (See Section X).

Programs punched onto paper tape in the above manner, or produced by the PUNCH command, may be input to the system through the paper tape reader after typing the TAPE command. If your system has a paper tape reader, you may be able to use the LOAD command to enter your program. (See Section X).

The paper tape punch may also be used off-line to prepare a list of data that will be input in a BASIC program with INPUT, LINPUT, or ENTER statements. Using the same procedure described above for preparing a BASIC program, type each line of data as usual, terminating each line with X-OFF (S^c), *return linefeed.*

Note: For further discussions of data format, refer to the INPUT, LINPUT, and ENTER statements. The number of characters permitted in a line of data depends on your system configuration.

To use the data in a BASIC program, insert the data tape in the paper tape reader and set the reader to AUTO. The data will then automatically be read in at INPUT, LINPUT, and ENTER statements when the program is run.

USER COMMAND ERROR MESSAGES

Specific command error messages are listed below along with the commands which generate them. The message ILLEGAL FORMAT is produced by all commands when their parameter string is improperly constructed. The messages listed here usually indicate a syntactically correct command which is rejected for the reason indicated.

APPEND

ENTRY IS A FILE
EXECUTE ONLY
NO COMMON AREA ALLOWED
NO SUCH PROGRAM
PROGRAM TOO LARGE
SEMI-COMPILED PROGRAM
SEQUENCE NUMBER OVERLAP
UNABLE TO RETRIEVE FROM LIBRARY

CREATE

CONFLICTING ACTION BY OTHER USER
DUPLICATE ENTRY
ILLEGAL PARAMETER
LIBRARY SPACE FULL
SYSTEM OVERLOAD
UNSUCCESSFUL, PURGE AND RETRY

CSAVE

DUPLICATE ENTRY
LIBRARY SPACE FULL
NO PROGRAM
NO PROGRAM NAME
OUT OF STORAGE
RUN ONLY
SYSTEM OVERLOAD
UNSUCCESSFUL, TRY AGAIN

(Since execution of this command requires some of the same processing as the RUN command, it may also produce many of the execution errors listed later in this appendix.)

DELETE

NOTHING DELETED

EXECUTE

ENTRY IS A FILE
NO SUCH PROGRAM
PROGRAM TOO LARGE
UNABLE TO RETRIEVE FROM LIBRARY

(Since this command includes the processing of a RUN command, it can also produce any of the execution errors listed later in this appendix.)

FILE

CONFLICTING ACTION BY OTHER USER
DEVICE UNAVAILABLE
DUPLICATE ENTRY
ILLEGAL PARAMETER
LIBRARY SPACE FULL
RECORD SIZE TOO LARGE
SYSTEM OVERLOAD
UNSUCCESSFUL, PURGE AND RETRY

GET

ENTRY IS A FILE
EXECUTE ONLY
NO SUCH PROGRAM
PROGRAM TOO LARGE
UNABLE TO RETRIEVE FROM LIBRARY

HELLO

ILLEGAL ACCESS
NO TIME LEFT

LIST

CHARACTERS AFTER COMMAND END
PROGRAM BAD
RUN ONLY

LOAD

ILLEGAL FORMAT
FILE WRITE ONLY
FILE READ ONLY OR NOT ASCII (Only "NOT ASCII" applies to LOAD)
FILE/DEVICE BUSY OR NOT PRESENT
DEVICE UNAVAILABLE
RECORD SIZE TOO LARGE
OUT OF STORAGE

LOCK

NO SUCH ENTRY

MESSAGE

CONSOLE BUSY

MWA

ASCII FILE, NOT ALLOWED
ENTRY IS A PROGRAM
INSUFFICIENT CAPABILITY
NO SUCH ENTRY
NO SUCH ID

NAME

ONLY 6 CHARACTERS ACCEPTED

PRIVATE

NO SUCH ENTRY

PROTECT

NO SUCH ENTRY

PUNCH

ASCII FILE NOT PERMITTED
ASCII FILE REQUIRED
CHARACTERS AFTER COMMAND END
RUN ONLY
PROGRAM BAD

PURGE

FILE IN USE
NO SUCH ENTRY

RENUMBER

BAD PARAMETER
SEQUENCE NUMBER OVERFLOW/OVERLAP

RUN

(See the list of execution errors later in this appendix.)

SAVE

DUPLICATE ENTRY
LIBRARY SPACE FULL
NO PROGRAM
NO PROGRAM NAME
OUT OF STORAGE
RUN ONLY
SYSTEM OVERLOAD
UNSUCCESSFUL, TRY AGAIN

SWA

ENTRY IS A PROGRAM
NO SUCH ENTRY

UNRESTRICT

NO SUCH ENTRY

Use of the *OUT=file name* construct in one of the above commands can also produce the following messages

DEVICE UNAVAILABLE
END OF FILE
FILE/DEVICE BUSY OR NOT PRESENT
FILE READ ONLY OR NOT ASCII
RECORD SIZE TOO LARGE

If users other than account A000 attempt to use certain commands reserved for the system manager they will receive the message

PRIVILEGED COMMAND

If a program entered under control of the TAPE command contained errors, execution of the next command entered will be replaced by printing of the error messages followed by

LAST INPUT IGNORED, RETYPE IT

LANGUAGE PROCESSOR ERROR MESSAGES

The following messages are output by the BASIC language processor to indicate errors or possible errors in users' BASIC programs.

SYNTAX ERRORS

One of the following error messages will be produced after entry of a BASIC statement with incorrect syntax. The incorrect line is rejected rather than added to the BASIC program unless the message is OVER/UNDERFLOW – WARNING ONLY.

The messages are listed in alphabetic order.

CHARACTERS AFTER STATEMENT END
EXTRANEIOUS LIST DELIMITER
ILLEGAL EXPONENT
ILLEGAL OPERAND AFTER 'USING'
ILLEGAL OR MISSING INTEGER
ILLEGAL READ VARIABLE
ILLEGAL SYMBOL FOLLOWING 'MAT'
MATRIX CANNOT BE ON BOTH SIDES
MISSING ASSIGNMENT OPERATOR
MISSING LEFT PARENTHESIS
MISSING OR BAD ARRAY VARIABLE
MISSING OR BAD FILE REFERENCE
MISSING OR BAD FUNCTION NAME
MISSING OR BAD LIST DELIMITER
MISSING OR BAD SIMPLE VARIABLE
MISSING OR BAD STRING OPERAND
MISSING OR ILLEGAL DATA ITEM
MISSING OR ILLEGAL 'OF'
MISSING OR ILLEGAL 'STEP'
MISSING OR ILLEGAL SUBSCRIPT
MISSING OR ILLEGAL 'THEN'
MISSING OR ILLEGAL 'TO'
MISSING RELATIONAL OPERATOR
MISSING RIGHT PARENTHESIS
NO CLOSING QUOTE
NO LEGAL BINARY OPERATOR FOUND
NO '*' AFTER RIGHT PARENTHESIS
OUT OF STORAGE
OVERFLOW/UNDERFLOW – WARNING ONLY
PARAMETER NOT STRING VARIABLE
'PRINT' MUST PRECEDE 'USING'
SIGN WITHOUT NUMBER
STATEMENT HAS EXCESSIVE LENGTH
STRING VARIABLE NOT LEGAL HERE
UNDECIPHERABLE OPERAND
VARIABLE MISSING OR WRONG TYPE
255 CHARACTERS MAX FOR STRING

EXECUTION ERRORS

Erroneous conditions discovered during execution of a program cause the program to stop execution and also result in one of the messages below. These errors are all diagnosed following a RUN or EXECUTE command. They may include errors discovered during compilation. Except for the compilation errors, all the errors are assigned a number that may be recovered through execution of the SYS(0) function. The programming, format, and ASCII file errors can be trapped by the IF ERROR statement that interrupts the flow of execution and prevents the message from being sent to the user terminal.

Certain errors only result in a warning message and do not terminate execution of the program. IF ERROR does not interrupt program execution for such errors but does prevent display of the error message.

COMPILE TIME ERRORS. The errors listed below are not assigned a number. These errors and those starred errors among the programming errors cannot be trapped by the IF ERROR statement.

Error Message	Cause and Corrective Action
ARRAY OF UNKNOWN DIMENSIONS	Undimensioned array; dimension with DIM statement.
ARRAY TOO LARGE	Array dimensioned with more than 5000 elements; decrease dimensions in DIM.
BAD FORMAT OR ILLEGAL NAME	FILES statement incorrectly formatted or contains incorrectly formatted file name; correct FILES statement.
CHARACTERS AFTER COMMAND END	Characters follow last parameter allowed in a RUN, LIST, or PUNCH command; correct command.
DIMENSIONS NOT COMPATIBLE	Number of dimensions (one or two) in DIM statement differs from number of dimensions specified in array reference; redimension array.
FUNCTION DEFINED TWICE	More than one user-defined function defined with the same name; assign new name to function.
LAST STATEMENT NOT 'END'	END statement is missing; supply the statement.
MISSING OR PROTECTED FILE	File referenced is not in library or is not accessible; create file if missing.

NEXT WITHOUT MATCHING FOR	NEXT statement specifies variable not defined in FOR statement; check variable and include missing FOR statement if necessary. Also returned for improper nesting of FOR ... NEXT statements; correct and rerun.
OUT OF STORAGE	No more room in user workspace; try to shorten program length.
Error Message	Cause and Corrective Action
SAME FOR-VARIABLE NESTED	A nested FOR statement uses the same for-variable as FOR statement at higher level; check code and change variable.
UNDEFINED FUNCTION	Function references a function that has not been defined; check function name, define function if necessary.
UNDEFINED STATEMENT REFERENCE	Referenced statement does not exist; check statement number, include statement if necessary.
UNMATCHED FOR	FOR statement does not have matching NEXT statement; check for variable and include NEXT statement if missing.
VARIABLE DIMENSIONED TWICE	The same variable has been dimensioned twice; check variable and delete one DIM or COM specification if necessary.

PROGRAMMING ERRORS. The following errors are assigned a number that can be recovered through execution of SYS(0) if an IF ERROR statement has been specified. With the exception of the starred messages, IF ERROR causes the flow of execution to be altered when one of the following errors occurs. Note that the line where the error occurred can be retrieved by executing SYS(1) when IF ERROR was specified.

Error Number	Error Message	Cause and Corrective Action
1	GOSUBS NESTED TWENTY DEEP	No more than 20 GOSUB statements without an intervening RETURN statement can be nested. This error may occur because IF END or IF ERROR causes exit from subroutine without executing RETURN.
2	RETURN WITH NO PRIOR GOSUB	A RETURN statement encountered with no prior GOSUB in the sequence of execution. This error may occur because program has no STOP statement prior to subroutine.

Error Messages

Error Number	Error Message	Cause and Corrective Action
3	SUBSCRIPT OUT OF BOUNDS	An array element specified by a subscript greater than the array dimension.
4	NEGATIVE STRING LENGTH	Second subscript in substring specification is less than the first.
5	NON-CONTIGUOUS STRING CREATED	Values assigned to string variable must be contiguous from left to right; e.g., the following code generates this error: DIM A\$(20) A\$(1,5)="ABCDE" A\$(10,12)="JKL" positions 6-9 of A\$ are undefined.
6	STRING OVERFLOW	Number of characters in string literal is greater than dimensions of string variable, or string entered in response to LINPUT, ENTER, INPUT, or READ was larger than dimensions allow.
7	OUT OF DATA	READ statement has come to the end of data in the DATA statements.
8	DATA OF WRONG TYPE	Data in a DATA statement or file is a different type (string or numeric) than corresponding variable.
9	UNDEFINED VALUE ACCESSED	Variable referenced to which no value has been assigned.
10	MATRIX NOT SQUARE	MAT...IDN statement requires an array with the same number of rows as columns.
11	REDIMENSIONED ARRAY TOO LARGE	New array dimensions greater than physical dimensions of array specified by DIM.
12	NEARLY SINGULAR MATRIX	Array cannot be inverted (MAT...INV); determinant of array is nearly zero.
13	LOG OF NEGATIVE ARGUMENT	Argument of LOG function not a positive value.
14	SQR OF NEGATIVE ARGUMENT	Argument of SQR function not a positive value.
15	ZERO TO ZERO POWER	Exponent of a zero value is itself zero.

Error Number	Error Message	Cause and Corrective Action
16	NEGATIVE NUMBER TO REAL POWER	Exponent is not an integral value and number is negative (e.g., $-2^{1.3}$).
17	TRIG FUNCTION ARGUMENT OUT OF RANGE	Argument of ATN, COS, SIN, or TAN function is outside allowed range; check function descriptions for range and correct.
18	OVER/UNDERFLOW — WARNING ONLY	Result of calculation has produced a number either too large or too small.
*19	LAST INPUT IGNORED, RETYPE IT	Issued after tape mode errors to indicate that last command typed was ignored.
*20	TOO MANY FILES STATEMENTS	Compilation error; more than 4 FILES statements.
21	NON-EXISTENT FILE REQUESTED	Referenced file number not opened by ASSIGN or position not named in FILES statement.
22	WRITE TRIED ON READ ONLY FILE	Attempt to print data to a file from which data may only be read.
23	END-OF-FILE/ END-OF-RECORD	Attempt to sequentially print or read beyond physical end-of-file, or attempt to direct print or read beyond physical end of record or file.
24	STATEMENT NOT IMAGE	Statement referenced in PRINT USING not an IMAGE statement.
*25	NON-EXISTENT PROGRAM REQUESTED	Program named in CHAIN statement not in library, not accessible, or specified incorrectly; may be trapped if return variable specified.
*26	CHAIN REQUEST IS A FILE	Program name, not a file name, must be specified in CHAIN statement; may be trapped if return variable specified.
*27	PROGRAM CHAINED TO IS TOO LARGE	Program named in CHAIN statement is too large for user work space; may be trapped if return variable specified.
*28	COM STATEMENT OUT OF ORDER	COM statements must be lowest numbered statements in program; CHAIN error.

Error Messages

Error Number	Error Message	Cause and Corrective Action
29	ARGUMENT OUT OF RANGE	Function argument is outside legal range.
*30	(not currently used)	
31	BAD FILE READ	Fatal disc error trying to read disc block; contact your operator who may fix the problem or else will call the HP representative.
32	BAD FILE WRITE DETECTED	Fatal disc error trying to write disc block; contact your operator who may fix the problem or else will call the HP representative.
*33	CAN'T READ PROGRAM CHAINED TO	Program named in CHAIN cannot be accessed; may be trapped if return variable specified.
*34	NO ACCESS ALLOWED	Specified program cannot be executed; may be trapped if return variable specified.
*35	PROGRAM BAD	Internal system problems; contact your operator who may fix the problem or else will call your HP representative.
*36	STATEMENT NUMBER OUT OF BOUNDS	Specified statement number does not exist in program chained to; may be trapped if return variable specified.
37	NO ACCESS AT THIS TIME	Device requested in FILES or ASSIGN statement is busy; try again later.
38	ASCII FILE NOT PERMITTED	Attempt to use LOCK, UNLOCK, ADVANCE, UPDATE, REC, ITM, etc., with an ASCII file.
39	RETURN VARIABLE NEEDED	Return variable must be specified in a LOCK because at least one other file is locked.
40	ASCII FILE REQUIRED	LINPUT, PRINT#USING, etc., require an ASCII file; attempt made to use BASIC Formatted file.
41	MISSING OR INVALID COMMAND	Command omitted or not specified correctly or not allowed in SYSTEM statement.

Error Number	Error Message	Cause and Corrective Action
42	READ TRIED ON WRITE-ONLY FILE	Attempt to read from a file to which you only have write access (e.g., LP, PP, etc.)
*43	(not currently used)	
44	RECORD NUMBER NOT ALLOWED	Direct access attempted on ASCII file or file # 0 (your terminal).



FORMAT ERRORS. The following errors, like programming errors, terminate execution unless trapped through the IF ERROR statement. These errors are caused by incorrectly specified formats in IMAGE, PRINT USING, or MAT PRINT USING statements.

Error Number	Error Message	Cause and Corrective Action
100	MISSING FORMAT SPECIFICATION	No format specification in PRINT USING or IMAGE statement.
101	ILLEGAL OR MISSING DELIMITER	Delimiter not comma or slash or is missing.
102	NO CLOSING QUOTE	Format string not terminated by quote.
103	BAD CHARACTER AFTER REPLICATOR	Integer indicating repetition may only precede characters A, D, or X.
104	REPLICATOR ZERO OR TOO LARGE	Replicator must be in range 1 through 255.
105	MULTIPLE SIGNS	Only one sign allowed in a numeric specification.
106	MULTIPLE DECIMAL POINTS	Only one decimal point allowed in a numeric specification.
107	BAD FLOATING SPECIFICATION	Invalid format for a floating point specification in format string (e.g., DD.DDEEE).
108	ILLEGAL CHARACTER IN FORMAT	Character other than A,D,S,E,X,/, a decimal point, or quoted string not allowed in format.
109	ILLEGAL FORMAT FOR STRING	Specification for a number associated with a string variable (e.g., PRINT USING "DD";A\$).

Error Messages

Error Number	Error Message	Cause and Corrective Action
110	MISSING RIGHT PARENTHESIS	Right parenthesis missing in grouped format specification.
111	MISSING REPLICATOR	Group must be preceded by integer between 1 and 255 used as repetition factor.
112	TOO MANY PARENTHESSES LEVELS	Groups may be nested only two deep.
113	MISSING LEFT PARENTHESIS	Left parenthesis missing in grouped format specification.
114	ILLEGAL FORMAT FOR NUMBER	Format specification is for a string, not a number (e.g., PRINT USING "AAA";A).

EXECUTION WARNINGS. Warning errors occur under the same circumstances as other execution errors. These errors, however, represent conditions that are either recoverable or simply questionable practices and do not terminate execution nor are they trapped by IF ERROR. The associated error numbers may be recovered through execution of SYS(0). When IF ERROR has been specified, the message is not printed.

Error Number	Error Message	Cause and Corrective Action
200	BAD INPUT, RETYPE FROM ITEM	Bad format in item read by INPUT statement; retype items starting with specified item number.
201	LOG OF ZERO – WARNING ONLY	Argument of LOG function evaluates to zero; function value is set to -10^{38} .
202	ZERO TO NEGATIVE POWER – WARNING	Zero raised to negative power (e.g., 0^{*-1}); value set to 10^{38} .
203	DIVIDE BY ZERO – WARNING ONLY	Zero divisor (e.g., $1/0$); value set to 10^{38} .
204	EXP OVERFLOW – WARNING ONLY	EXP(x) executed where x greater than 85; value set to 10^{38} .
205	OVERFLOW – WARNING ONLY	Magnitude of result of calculation greater than 10^{38} .
206	UNDERFLOW – WARNING ONLY	Magnitude of result of calculation less than 10^{-38} but not zero.

207	EXTRA INPUT — WARNING ONLY	More items typed in response to INPUT statement than were requested.
208	OVER/UNDERFLOW — WARNING ONLY	Overflow or underflow occurred when reading or writing an ASCII file.
209	TRANSMISSION ERROR. REENTER	Characters entered in response to INPUT or LINPUT statement were either lost or garbled.

ASCII FILE ERRORS. The following errors result from attempts to access ASCII files. Like other execution errors, they cause the program to terminate unless trapped through the IF ERROR statement. The error number is recoverable through SYS(0).

Error Number	Error Message	Cause and Corrective Action
300	NO DATA AVAILABLE ON RJE OR LT	1.) No data found when reading from LT device, and EOF not reached. 2.) No data available from JL or JT device because RJE buffer is full. Unless trapped by IF ERROR, system tries again until data available.
301	DEVICE NOT READY	Device associated with ASCII file is not ready; ready device and restart program.
302	DEVICE ERROR	Card reader improperly read a card; replace card in hopper, and make device ready.
303	ATTENTION NEEDED	Operator attention is required by ASCII file device; invalid Hollerith code read on card reader, or magnetic tape off-line or not at load point when device first accessed.
304	READ/WRITE FAILURE	Power failed on a non-recoverable device (MT, PR) or bad parity read from PR, or no write ring in MT when attempt made to write, etc.

TERMINAL INTERFACE

APPENDIX

D

Ten types of user terminals can be connected to the 2000 System. Nine generate ASCII code and one generates CALL 360 or PTTC/EBDC (non-ASCII) code.

The following terminals generate ASCII code:

- HP 2749 Teleprinter Terminal; ASR-33 or ASR-38 Teleprinter Terminals
- HP 2640A or HP 2644A Interactive Display Terminals
- HP 2600A CRT Terminal
- HP 2762A Data Communications Terminal; strapped for ECHO-PLEX
- GE TermiNet 300 Data Communications Terminal, Model B (10/15/30 cps) with Paper Tape Reader/Punch, Option 2; strapped for ECHO-PLEX
- GE TermiNet 1200 Data Communications Terminal
- ASR-37 Teleprinter Terminal with Paper Tape Reader/Punch
- GE TermiNet 30 Data Communications Terminal
- Texas Instruments Silent 700
- Execuport 300 Data Communications Transceiver Terminal
- HP 3071A Transaction Terminal

Notes: If the ASR-37 is equipped with the Shift Out (SO) feature, SO must be disabled because the 2000 System does not allow use of this feature. The HP 2760A Optical Mark Reader can be connected in parallel with any of the terminals listed, providing a useful terminal-reader combination.

The following user terminal generates non-ASCII code:

- IBM 2741 Communication Terminal

Note: The IBM 2741 terminal must be connected to the system over telephone lines. In addition, the terminal must be equipped with the following features:

1. Interrupt, Receive (IBM #4708) and Transmit (IBM #7900) associated with the terminal's ATTN key.
2. Dial-Up (IBM #3255) to enable system connection through a 103A modem or acoustic coupler.

In order to log on to the 2741, press the ATTN key rather than carriage return and linefeed. If your 2741 generates PTTC/EBDC code, the PLEASE LOG IN message will be garbled since the system only detects the code type when the H in the HELLO command is typed.

In order to get the PLEASE LOG IN message when using a terminal without a linefeed key, you may either:

- press the carriage return key, followed by a control J (J^C)
- if the terminal has an AUTO LF key, turn the key on and press carriage return. If the terminal is operating under half duplex, return the AUTO LF key to off position after the PLEASE LOG IN message.

Any terminal operating under half duplex with an AUTO LF key should be operated with the auto linefeed feature off.

Note: Although cursor, form feed, horizontal and vertical tabulation, and various special function keys are provided on specific types of user terminals, these capabilities are only supported by the 2000 System for type 2 terminals operating in page mode. Some of these operations may be requested from the keyboard, but results are unpredictable. Features provided by 2000 BASIC, such as the TAB, SPA, and LIN functions, and the PRINT and PRINT USING statements, should be used to control output format. These functions and statements are described in other sections of this manual. Terminals equipped with automatic linefeed after carriage return or on end of line may cause unpredictable results.

A terminal type (0-8) entered with the HELLO command is associated with each of the nine ASCII code terminals. Type 9 is associated with non-ASCII generating terminals; this type need not be specified, but is returned if requested by the SYS function. Each of the terminal types is shown below in a matrix (Table D-1) that associates type with typical terminals and gives the delay following a carriage return or linefeed, the terminal backspace character, and the backspace indicator for terminals that do not physically backspace.

The delay is specified as two numbers that indicate the number of fill characters sent to the terminal after a carriage return (first number) and after a linefeed (second number). The pairs of numbers are shown for the baud rates to which these terminals may be set.

When you backspace with H^C the 2000 system moves the print head or cursor back one space unless the terminal is incapable of physically backspacing, in which case, a left arrow (←) or underline (—) is displayed for each character deleted. In the matrix, *none* indicates that the terminal allows physical backspacing. The backspace key or the sequence ϕ ch can be used to delete characters on the IBM 2741 terminal (type 9).

Table D-1. Matrix of Terminal Types

TERMINAL TYPE	REPRESENTATIVE TERMINAL	DELAYS BY BAUD RATE						BACKSPACE CHARACTER	PRINTED INDICATOR
		110	150	300	600	1200	2400		
0	HP 2749A ASR 33 ASR 38 HP 3071A	1,0	0,0	0,0	0,0	0,0	0,0	H ^c	← or -
1	HP 2640A HP 2644A	0,0	0,0	0,0	0,0	0,0	0,0	H ^c	none
2	HP 2640A HP 2644A (page mode)	0,0	0,0	0,0	0,0	0,0	0,0	H ^c	none
3	HP 2600A	0,0	0,0	0,0	0,0	4,0	4,0	H ^c	none
4	HP 2762A GE TermiNet 300 GE TermiNet 1200	0,3	0,5	0,9	0,0	0,36	0,0	H ^c	none
5	ASR 37	0,0	1,0	0,0	0,0	0,0	0,0	H ^c	←
6	Terminet 30	4,0	0,0	12,0	0,0	0,0	0,0	H ^c	←
7	TI Silent 700	2,1	4,1	6,1	0,0	0,0	0,0	H ^c	none
8	Execuport 300	0,0	2,0	4,0	0,0	0,0	0,0	H ^c	none
9	IBM 2741	-	-	-	-	-	-	BACKSPACE (⌘CH)	none

Types 4 and 5. These terminals support a paper tape reader. Whenever an X-OFF character (S^c) is detected, it is treated like a carriage return. Subsequent characters up to and including the next actual carriage return are ignored. This allows proper operation of the paper tape readers on these terminals.

Note: Please refer to your Owner's Manual for the HP 2640 or HP 2644 in order to fully understand the following discussion of the use of these terminals with the HP 2000 system. It is also necessary to refer to your Owner's Manual for the HP 3071A in order to fully understand its use with the 2000 system.

Types 1 and 2. Terminals of this type always reply to a E^c (ENQ) character with a F^c (ACK) character. This ENQ/ACK sequence indicates a readiness to accept more data and obviates the need for fill characters after output. If you have successfully logged on specifying a type 1 or 2 terminal, the 2000 system sends an ENQ character after every 80 characters of output and discards the ACK returned by the terminal. It also discards the ACK returned as a result of a user typing E^c or ENQ that is echoed at the terminal. If, however, you type E^c before the HELLO command has been executed, the system does not know your terminal type and does not discard the ACK returned by the terminal. Because of this, E^c should not be used in a password.

If you accidentally specify type 1 or 2 from a terminal of another type, the ENQ/ACK sequence is discarded so that terminals not using ENQ/ACK can operate correctly. Other problems may still occur since there is no delay during output and a F^C entered immediately following E^C is discarded. For these reasons, it is important to specify your terminal type correctly.

On terminal types 1 or 2, care must be taken when performing tape cartridge operations. The 2000 system sets a five second time limit following the output of each ENQ and after that time has elapsed it assumes an ACK was returned by the terminal and continues program execution. Because tape cartridge operations are often longer than the HP 2000 time limit and because the terminal only sends an ACK when the cartridge operation is complete, characters sent immediately after the characters that initiate the cartridge operation may be ignored. To avoid this possibility, always follow a statement that affects cartridges with an ENTER or LINPUT statement. The terminal returns a status character to the string variable specified in such a statement. To illustrate:

```
10 PRINT '27"&p2uOC"      escape sequence to rewind right cartridge
20 LINPUT A$             A$ is set to the status code
```

The status code returned in A\$ has one of the following values:

```
S - operation was successful
F - operation failed
U - user interrupted operation
```

Writing to a tape cartridge with a byte count on a terminal of type 1 or 2 may take additional time if any of the bytes is E^C (ENQ). Because of the way the 2000 System handles the ENQ/ACK sequence, the print operation must wait until the system times out following each E^C in the byte string. Also, if more than 80 characters are output in the byte string, the system-generated ENQ is written to the tape.

Note: Refer to the HP 2640 or HP 2644 Owner's manual for information on operating these terminals in page block mode before reading the next paragraphs.

Type 2. When an HP 2640 or HP 2644 terminal is to be operated in page block mode, type 2 should be specified at log on so that the HP 2000 system will recognize as special characters, the RS, or record separator, (^^C) and DC2 (R^C). For type 2 terminals, a DC2 or an RS is treated as a carriage return.

After entering data in block mode at the terminal, the terminal user presses the ENTER key to send this data to the HP 2000. ENTER sends a DC2 character to the system that is treated as a carriage return by HP 2000 and that should be input to the user's program with a LINPUT or ENTER statement. A block of data may then be read by a second LINPUT or ENTER statement. Since the 2000 LINPUT and ENTER statements only accept data in blocks of 255 characters or less, a record separator (RS) must be inserted in any data that contains more than 255 consecutive characters. The following sample program illustrates the sequence of commands to set up for input from a 5-character unprotected field in block mode.

In this example, the escape key is represented by '27. The escape sequences all control the terminal; in lines 50 through 70, they turn format off, home the cursor, clear the display, disable the keyboard, set up and display the form, and then turn on format mode. After data has been read, they are used in line 140 to turn format off, home the cursor, and clear display following input, and then in line 160 to enable the keyboard.

The escape sequences should be entered continuously with no intervening linefeed. Either a LIN(0) function or a terminating semicolon may be used; LIN(0) should be specified at intervals to insure the character count never reaches 72 causing the system to issue a linefeed.

```

10 DIM A$(255)
20 REM ... SET UP FOR DATA INPUT IN BLOCK MODE
30 ENTER 5,A,A$           5-second delay to allow user to press
40 SYSTEM A,"ECHO-OFF"    block key. Suppress echo at terminal
50 PRINT LIN(0);'27"X"'27"H"'27"J"'27"C"
60 PRINT "DATA "'27"&tb"'27"[           "'27"]"'27"&dw";
70 PRINT '27"W";
80 REM ... GET THE DATA
90 PRINT '27"b";           Enable keyboard, wait for DC2
100 LINPUT A$             Pressing ENTER sends DC2 to A$
110 PRINT                 '27"C"'27"H"       Disable keyboard, home cursor
120 LINPUT A$             Read data into A$
130 REM ... CLEAN UP AFTER DATA INPUT
140 PRINT '27"X"'27"H"'27"J";
150 SYSTEM A,"ECHO-ON"    Enable echo
160 PRINT '27"b";         Enable keyboard entry
170 PRINT A$
180 END

```

If the data entered contains more than 255 consecutive characters, you must include an RS character after blocks of data so that no block contains more than 255 characters. The RS characters should be placed on the screen after ENTER is pressed and DC2 sent to the system (line 100) but before data is read from the terminal (line 120). To place an RS character on the screen, write the following code:

```

112 PRINT '27"&a79C"      move cursor to desired RS position
114 PRINT '27" ";        start an unprotected field
116 PRINT '30;           print the record separator (RS)
118 PRINT '27" ";        end an unprotected field

```

When the system receives the RS, it assumes that the data transfer is over.

After reading the first block of data in line 120 an escape sequence must be sent to the terminal to enable block transfer from the computer before each subsequent block of code can be read. The following code would be inserted to read one additional block from the terminal:

```

122 PRINT '27"d"
124 LINPUT A$

```

If data is contained in more than one unprotected field, the terminal inserts a unit separator (US) between the data transferred from each field. If a record separator (RS) terminates a field, it replaces the unit separator. The unit separator code (-^c) can be useful to the BASIC program to distinguish one data field from another.

Refer to the Owner's Manual for either the HP 2640 or the HP 2644 terminals for more information on operating in page mode or for general information about the terminals.

IBM 2741 COMMUNICATIONS TERMINAL INTERFACE

Because the IBM 2741 terminal generates non-ASCII code, special consideration must be given to the representation of several ASCII characters and functions which are not available in the 2741 character set.

For input from a 2741 terminal, these characters (and some of the functions) are simulated by entry of a two-character code. The first character of this code is the cent symbol (¢). The cent symbol is followed by one of several alphanumeric or special characters to compose a unique code representing one ASCII character or function.

On input from a 2741 terminal, the two-character code is translated into the internal ASCII code. On output to a 2741 terminal, ASCII code is translated into the appropriate two-character representation.

The IBM 2741 Communications Terminal must be equipped with the interrupt feature associated with the *ATTN* key. This key represents the *break* function; it is used to terminate program or command execution.

An IBM Selectric 2741 with a type ball numbered 1167087 uses CALL/360 code; EBCD code is used with an IBM PTTC/EBCD 2741 having a type ball with part number 1167963.

Any CALL/360 or PTTC/EBCD characters that do not have an equivalent ASCII character are ignored on input.

Table D-2 shows 2741 terminal representation of ASCII characters and functions.

Table D-2. IBM 2741 ASCII Character Simulation

ASCII Graphic Control Character	IBM 2741 Character Representation		User Terminal Function	IBM 2741 Character Representation	
	CALL/360	PTTC/EBCD		CALL/360	PTTC/EBCD
	¢(¢(<i>control</i> ^① <i>break</i>	¢C	¢C
\	¢/	¢/		ATTN key	ATTN key
	¢)	¢)			
^	¢↑	¢A			
~	¢'	¢'			
0	¢0	¢0			
S	¢S	¢S			
T	¢T	¢T			
—	—	—			
ESC	¢E	¢E			
FS	¢F	¢F			
GS	¢G	¢G			
RS	¢R	¢R			
US	¢U	¢U			

^① Code must be followed by an appropriate alphabetic character; otherwise, it is ignored.

Examples:

	ASCII Character	IBM 2741 Equivalent
To delete a line of input	Control X	<i>¢CX return</i>
To delete a character	Control H	<i>¢CH return</i>

ADDITIONAL LIBRARY FEATURES

APPENDIX

E

The system operator has several program and file movement capabilities of which the user should be aware. In addition, there are some commands available to the operator at system startup time which allow storage and retrieval of programs and files on magnetic tape. These operator commands, and their functions, are listed here. The discussions of operator commands that follow assume that you are familiar with the library and security structure of the system (refer to Section VIII).

BESTOW

This command enables the operator to transfer a program or file, or an entire library from one account to another. Individual library entries may be transferred any time the system is running. Entire libraries may be transferred only when no users are logged on and the system is running. Library entries which are PRIVATE or LOCKED will not be transferred. UNRESTRICTED or PROTECTED entries remain UNRESTRICTED or PROTECTED. An MWA, (Multiple Write Access) file remains MWA only if the new library's idcode has the MWA capability.

COPY

This command is used to make a duplicate copy of any user program or file in the library of any other user (or the same user). The copy may be given a new name at the time it is created. A program or a file of fewer than 200 blocks may be transferred any time the system is running. Files of greater than 200 blocks may be copied only when no users are logged on and the system is running. Entries which are PRIVATE or LOCKED will not be transferred. UNRESTRICTED or PROTECTED entries remain UNRESTRICTED or PROTECTED. An MWA (Multiple Write Access) file remains MWA only if the new library's idcode has the MWA capability.

LOAD

At system startup time, the system operator may use the **LOAD** command to load selected programs and files, entire user libraries, or all entries from magnetic tape. This tape must have been produced by a **DUMP**, **SLEEP**, or **HIBERNATE**. Entries already on the system will not be loaded. Program and file states (**UNRESTRICTED**, **LOCKED**, **PRIVATE**, or **PROTECTED**) will not be altered. An **MWA** (Multiple Write Access) file will remain **MWA** only if the library into which it is loaded has the **MWA** capability.

RESTORE

At system startup time, the system operator may use the **RESTORE** command to load selected programs and files, entire user libraries, or all entries from magnetic tape. This tape must have been produced by a **DUMP**, **SLEEP**, or **HIBERNATE**. The difference between **LOAD** and **RESTORE** is that **LOAD** does not replace existing entries; **RESTORE** does. Program and file states (**UNRESTRICTED**, **PROTECTED**, **LOCKED**, or **PRIVATE**) will not be altered. An **MWA** (Multiple Write Access) file will remain **MWA** only if the library into which it is loaded has the **MWA** capability.

DUMP

The **DUMP** command can be used by the system operator at system startup time to write selected user programs and files, entire user libraries, or all entries on a system to magnetic tape. Dump tapes are useful for archival storage, backup, or transferring entries between 2000 systems. The security and access states of programs and files are not altered when they are dumped.

This appendix contains a precise definition of the 2000 BASIC language. The descriptions listed here can be used to clarify the less formal definitions used in other parts of the manual. The syntactical grammar is described in a formal metalanguage derived from the Backus-Naur Form (BNF) of syntax definition.

The BNF notation consists of "productions" or syntax equations, each of which is in the following form:

$$\langle \text{syntactic entity} \rangle \quad := \langle \text{syntactic expression} \rangle$$

This can be read as "the entity on the left is composed of the ordered collection of one or more of the expressions on the right.

- If the entity has more than one expression, they are separated by a vertical bar "|". These expressions represent choices for any given expansion of the entity.
- Square brackets "[]" are used to enclose optional portions of expressions. The brackets can be nested (i.e., options can have options), and alternative options are expressed as a list of expressions separated by vertical bars, all enclosed within the brackets.
- Expressions will normally contain one or more entities. These can be expanded by substituting the right-hand side of the definitions for the entities.
- The syntax equations may be recursive (the entity on the left may appear in an expression used to define it.) When this occurs there is always at least one alternative which does not define the entity in terms of itself. This allows definitions where there are multiple occurrences of the same component.
- In some definitions the right-hand side of the equation will be a textual description rather than an expression.
- Numerals in definitions refer to notes which follow the formal definitions.

Formal Syntax for 2000 Basic

<letter>	::= A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
<digit>	::= 0 1 2 3 4 5 6 7 8 9
<signop>	::= + -
<relational operator>	::= < <= = > = > <> #
<integer>	::= <digit> <integer><digit>
<character string>	::= <character> <character string><character>
<character>	::= Any ASCII character ¹
<constant>	::= [<signop>] <numeric constants> <literal string>
<numeric constant>	::= <significant part> [<exponent>]
<significant part>	::= <integer> { <integer> }. [<integer>]
<exponent>	::= E [<signop>] <integer> ²
<literal string>	::= <quoted string> ' <integer> ³ [<quoted string>] <literal string> ' <integer> ³ [<quoted string>]
<quoted string>	::= " [<character string>] " ⁴
<numeric variable>	::= <numeric simple variable> <subscripted variable>
<numeric simple variable>	::= <letter> [<digit>]
<subscripted variable>	::= <array id> (<subscript> [, <subscript>]) ¹⁷
<array id>	::= <letter>
<subscript>	::= <numeric expression>
<return variable>	::= <numeric variable>
<string variable>	::= <string simple variable> { (<first character> [, <last character>]) } ¹⁷
<string simple variable>	::= <letter> [0 1] \$
<first character>	::= <numeric expression>
<last character>	::= <numeric expression>
<numeric expression>	::= <conjunction> <numeric expression> OR <conjunction>
<conjunction>	::= <relation> <conjunction> AND <relation>

<relation>	::= <minmax> <relation> <relational operator> <minmax>
<minmax>	::= <sum> <minmax> MIN <sum> <minmax> MAX <sum>
<sum>	::= <term> <sum> <signop> <term>
<term>	::= <subterm> <term> * <subterm> <term> / <subterm>
<subterm>	::= [<signop> <factor> NOT <factor>
<factor>	::= <primary> <factor> ↑ ¹⁸ <primary> <factor> ** <primary>
<primary>	::= <numeric constants> <numeric variable> <parameter> ⁵ <functional> (<numeric expression>)
<parameter>	::= <letter> <digit>]
<functional>	::= <function id> (<numeric expression>) <pre-defined function> (<numeric expression>) LEN(<source string>) NUM(<source string>) POS(<source string> , <source string>)
<function id>	::= FN <letter>
<pre-defined function>	::= ABS ATN BRK COS EXP INT ITM LOG REC RND SGN SIN SQR SYS TAN TIM TYP
<string expression>	::= <source string> CHR\$(<numeric expression>) UPS\$(<source string>)
<source string>	::= <literal string> <string variable>
<file reference>	::= <file expression> [, <record expression>]
<file expression>	::= ≠ <numeric expression>
<record expression>	::= <numeric expression>
<program>	::= <program statement> <program> <program statement> ⁶
<program statement>	::= <statement number> <BASIC statement>
<statement number>	::= <integer> ⁷
<BASIC statement>	::= <LET statement> <IF statement> <trap statement> <GOTO statement> <GOSUB statement> <RETURN statement> <FOR statement> <NEXT statement> <STOP statement> <END statement> <DATA statement> <READ statement>

<BASIC statement> (Continued)	<INPUT statement> <ENTER statement> <LINPUT statement> <RESTORE statement> <PRINT statement> <PRINT USING statement> <IMAGE statement> <REM statement> <DIM statement> <COM statement> <DEF statement> <FILES statement> <ASSIGN statement> <CHAIN statement> <CONVERT statement> <MAT statement> <CREATE statement> <PURGE statement> <ADVANCE statement> >UPDATE statement> <LOCK statement> <UNLOCK statement> <SYSTEM statement>
<LET statement>	::= [LET] <left part> <numeric expression> [LET] <destination string> = <string expression>
<left part>	::= <numeric variable> = <left part> <numeric variable> =
<destination string>	::= <string variable>
<IF statement>	::= IF <decision expression> THEN <statement number>
<decision expression>	::= <numeric expression> <string variable> <relational operator> <string expression>
<trap statement>	::= IF END <file expression> THEN <statement number> IF ERROR THEN <statement number>
<GOTO statement>	::= GOTO <statement number> GOTO <numeric expression> OF <statement number list>
<statement number list>	::= <statement number> <statement number list> , <statement number>
<GOSUB statement>	::= GOSUB <statement number> GOSUB <numeric expression> OF <statement number list>
<RETURN statement>	::= RETURN
<FOR statement>	::= FOR <for variable> = <initial value> TO <final value> [STEP <step size>]
<for variable>	::= <numeric simple variable>
<initial value>	::= <numeric expression>

<final value>	::= <numeric expression>
<step size>	::= <numeric expression>
<NEXT statement>	::= NEXT <for variable>
<STOP statement>	::= STOP
<END statement>	::= END
<DATA statement>	::= DATA <constant list>
<constant list>	::= <constant> <constant list> , <constant>
<READ statement>	::= READ [<file reference> ;] <read variable list> READ <file reference>
<read variable list>	::= <read variable> <read variable list> , <read variable>
<read variable>	::= <numeric variable> <destination string>
<INPUT statement>	::= INPUT <read variable list>
<ENTER statement>	::= ENTER # <numeric variable> ENTER [# <numeric variable> ,] <numeric expression> , <return variable> , <read variable>
<LINPUT statement>	::= LINPUT [<file expression> ;] <string variable>
<RESTORE statement>	::= RESTORE [<statement number>]
<PRINT statement>	::= <type statement> <file write statement>
<type statement>	::= PRINT [<print list> [, ;]]
<print list>	::= <print expression> <print list> , <print expression> ⁸ <print list> ; <print expression>
<print expression>	::= <numeric expression> <string expression> <print function>
<print function>	::= <print function id> (<numeric expression>)
<print function id>	::= LIN SPA TAB CTL ⁹
<file write statement>	::= PRINT <file reference> [; <write list>] ; END]
<write list>	::= <print list> [, [END]] ; [END]]
<PRINT USING statement>	::= PRINT USING <format part> [; <using list>] [<file PRINT USING statement>
<file PRINT USING statement>	::= PRINT <file reference> ; USING <format part> [; <using list>]

Formal Syntax for 2000 Basic

<format part>	::= <statement number> <string variable> “<format string>”
<format string>	::= [[<carriage control> ,] <format list>] ¹⁰
<carriage control>	::= + - ≠
<format list>	::= <format list element> / <replicator> (<format list>) ¹¹ <format list> , <format list element> <format list> / [<format list element>]
<format list element>	::= <literal string> ¹² <X list> <string specification> <integer specification> <fixed specification> <floating specification>
<X list>	::= <X part> <X list> <X part>
<X part>	::= [<replicator>] X
<replicator>	::= <integer> ¹³
<string specification>	::= [<X list>] <A part> <string specification> <A part> <string specification> <X list>
<A part>	::= [<replicator>] A
<integer specification>	::= [S] <num spec> ¹⁴
<num spec>	::= [<X list>] <D part> <num spec> <D part> <num spec> <X list>
<D part>	::= [<replicator>] D
<fixed specification>	::= [S] <num spec> . [<num spec>] ¹⁴ [S] . <num spec>
<floating specification>	::= <integer specification> E [<X list>] <fixed specification> E [<X list>]
<using list>	::= <using expression> <using list> , <using expression>
<using expression>	::= <numeric expression> <string variable> <print function>

<IMAGE statement>	::= IMAGE <format string>
<REM statement>	::= REM [<character string>
<DIM statement>	::= DIM <dimspec> <DIM statement> , <dimspec>
<dimspec>	::= <array id> (<bound> [, <bound>]) <string simple variable> (<bound>) ¹⁷
<bound>	::= <integer> ¹⁵
<COM statement>	::= COM <com list element> <COM statement> , <com list element>
<com list element>	::= <dimspec> <numeric simple variable> <string simple variable>
<DEF statement>	::= DEF <function id> (<parameter>) = <numeric expression>
<FILES statement>	::= FILES <file name designator> ¹⁰ <FILES statement> , <file name designator>
<file name designator>	::= [\$ *] <name> [, <account id>] *
<name>	::= A string of 1 to 6 letters and/or digits
<account id>	::= <letter> <digit> <digit> <digit>
<ASSIGN statement>	::= ASSIGN <file name> , <file number> , <return variable> [, <protection mask>] [, <restriction>] ASSIGN * , <file number> [, <return variable>]
<file name>	::= <source string>
<file number>	::= <numeric expression>
<protection mask>	::= <source string>
<restriction>	::= RR NR WR
<CHAIN statement>	::= CHAIN [<return variable> ,] <program name> [, <numeric expression>]
<program name>	::= <source string>
<CONVERT statement>	::= CONVERT <numeric expression> TO <string variable> CONVERT <source string> TO <numeric variable> [, <statement number>]

<MAT statement>	::= <MAT READ statement> <MAT INPUT statement> <MAT PRINT statement> <MAT PRINT USING statement> <file MAT PRINT USING statement> <MAT initialization statement> <MAT assignment statement>
<MAT READ statement>	::= MAT READ [<file reference> ;] <actual array> <MAT READ statement> , <actual array>
<actual array>	::= <array id> [<dimensions>]
<dimensions>	::= (<numeric expression> [, <numeric expression>])
<MAT INPUT statement>	::= MAT INPUT <actual array> <MAT INPUT statement> , <actual array>
<MAT PRINT statement>	::= <mat type statement> <mat file write statement>
<mat type statement>	::= MAT PRINT <mat print list> [, ;]
<mat print list>	::= <mat print expression> <mat print list> , <mat print expression> <mat print list> ; <mat print expression>
<mat print expression>	::= <array id> <print function>
<mat file write statement>	::= MAT PRINT <file reference> ; <mat print list> [, [END] ; [END]] MAT PRINT <file reference> ; END
<MAT PRINT USING statement>	::= MAT PRINT USING <format part> [; <mat print list>]
<file MAT PRINT USING statement>	::= MAT PRINT <file reference> ; USING <format part> [; <mat print list>]
<MAT initialization statement>	::= MAT <array id> = <initialization function> [<dimensions>]
<initialization function>	::= ZER CON IDN
<MAT assignment statement>	::= MAT <array id> [<signop> <array id>] MAT <array id> = <array id> * <array id> ¹⁶ MAT <array id> = TRN (<array id>) ¹⁶ MAT <array id> = INV (<array id>) MAT <array id> = (<numeric expression>) * <array id>
<CREATE statement>	::= CREATE <return variable> , <file name> , <file length> [, <record size>]
<file length>	::= <numeric expression>

<record size>	:= <numeric expression>
<PURGE statement>	:= PURGE <return variable> , <file name>
<ADVANCE statement>	:= ADVANCE <file expression> ; <skip count> , <return variable>
<skip count>	:= <numeric expression>
<UPDATE statement>	:= UPDATE <file expression> ; <numeric expression> UPDATE <file expression> ; <source string>
<LOCK statement>	:= LOCK <file expression> [, <return variable>]
<UNLOCK statement>	:= UNLOCK <file expression> [, <return variable>]
<SYSTEM statement>	:= SYSTEM <return variable> , <source string> ⁹ SYSTEM <string variable> , <source string>



NOTES:

1. The following ASCII characters are stripped by the system from terminal input and therefore cannot be entered directly: null, control-H, line-feed, carriage-return, X-OFF, control-X, and rubout.
2. Exponent integers are limited to exactly one or two digits.
3. The value of an integer used to supply a character within a literal string must lie between 0 and 255 inclusive.
4. The double quote character (") cannot appear within a quoted string.
5. A parameter primary can appear only in the defining expression of a DEF statement.
6. The last statement of a program must be an END statement.
7. A sequence number must lie between 1 and 9999 inclusive.
8. Print expressions which are literal strings need not be separated from preceding or following print expressions by semicolons or commas.
9. The CTL function and SYSTEM statement are requests for operating system services rather than a part of the BASIC language proper. They are included here as a convenient reference for their syntax. Users are explicitly cautioned that these constructs are not portable to any other Hewlett-Packard implementation of the BASIC language.
10. Any character string is accepted by the language grammar for a format string or a list of file names. The syntax of the string is checked when it is used during execution.
11. Groups in format lists can be nested only two levels deep.

12. In order to embed a literal string as a format list element into the quoted form of a format string (which itself is a literal string within the language grammar), the delimiting double quote characters of the literal string must be represented by means of the apostrophe convention (i.e., '34). The apostrophe convention is not needed or recognized when the format string occurs within an IMAGE statement or as the contents of a string variable. In no case can a double quote appear as a character within a literal string embedded in the format string referenced by a PRINT USING statement.
13. A replicator must lie between 1 and 255 inclusive.
14. An S can appear before, after, or between any two parts of a num spec except immediately following a replicator. Only one S can appear within one integer specification, fixed specification, or floating specification.
15. An array bound must lie between 1 and 9999 inclusive; a string variable bound must lie between 1 and 255 inclusive.
16. An array cannot be transposed into itself nor can it be both an operand and the result of a matrix multiplication.
17. Parentheses, (), and square brackets, [], are accepted interchangeably by the BASIC language.
18. A circumflex (^) may replace the up arrow (↑) on some terminals.

PROGRAMMING THE LINK TERMINAL

APPENDIX

G

This appendix contains short examples of programs using the link terminal devices. These programs illustrate techniques for controlling the various features of the link terminal, namely how to:

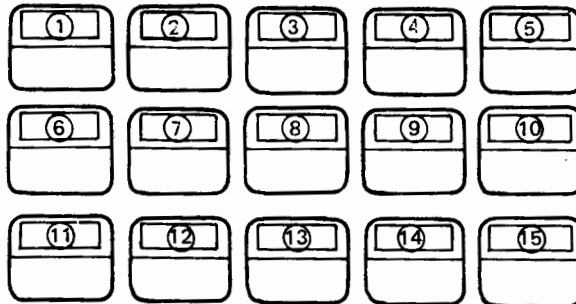
- manage the indicator lights
- make use of the special function keys
- control the display unit
- interface with the local HPIB.

In addition, a technique for writing BASIC programs which access multiple link terminals is described.

Note: Refer to your HP 3070A Owner's manual in order to fully understand the following discussion.

CONTROLLING THE INDICATOR LIGHTS

The following program shows how to switch on or off the indicator lights. It illustrates two formats of the arguments to the associated control function: strings representing numbers separated by spaces (– and , are also accepted separators) or integer variables. The convention for referencing the indicator lights is illustrated below.



```
FILE-TERM,LT
10 FILES TERM
20 PRINT #1;CTL(71),"1 2 3 4 5 6 7 8 9 10 11 12 13 15"
30 INPUT L
35 IF L=0 THEN 90
40 IF (L < -15) OR (L >15) THEN 70
50 PRINT #1;CTL(71),L
60 GOTO 30
70 PRINT "INCORRECT INDICATOR LIGHT NUMBER - RETYPE"
80 GOTO 30
90 END
```

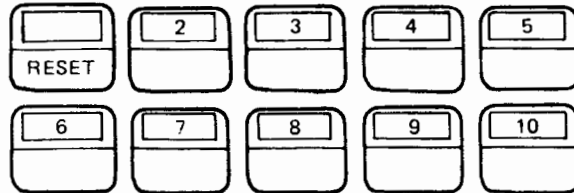
Programming the Link Terminal

- Comments: The ASCII file TERM is equated with a link terminal device.
- line 10: The link terminal is reset and all indicator lights are switched off.
- line 20: All 15 indicator lights are switched on.
- line 30: A numeric item is read from the user terminal and a check performed to make sure the number can be interpreted as an indicator light. If the number is outside the (- 15,+ 15) range an error message is printed on the user terminal and a new item requested. If the response is 0, the program ends in line 90.
- line 50: The indicator light referenced in line 30 is switched on if L is positive, or switched off if L negative.
- line 90: The program ends and all indicator lights are switched off as the Link Terminal is reset.

USING THE SPECIAL FUNCTION KEYS

In the following program selected special function keys are defined as terminators. A line of input is processed by a different section of the program depending on the terminating character.

The convention to follow in referring to the special function keys is illustrated below.



```
FILE-TERM,LT
10 FILES TERM
20 DIM A$(40)
30 PRINT #1;CTL(72),"2,3,4"
40 LINPUT #1;A$
50 L=LEN(A$)
60 IF L=0 THEN 900
70 B$=A$(L,L)
80 IF B$='16 THEN 200
90 IF B$='17 THEN 300
100 IF B$='18 THEN 400
110 PRINT "LINE ENDS WITH UNEXPECTED KEY - RETYPE"
120 GOTO 40
...
200 REM LINE TAGGED WITH SPECIAL FUNCTION KEY-2
...
300 REM LINE TAGGED WITH SPECIAL FUNCTION KEY-3
...
400 REM LINE TAGGED WITH SPECIAL FUNCTION KEY-4
...
900 END
```

- Comments: The link terminal device is equated with ASCII file TERM.
- line 10: The link terminal is reset and all special function keys are set to normal mode (as opposed to terminating mode).
- line 30: Special function keys numbered 2, 3 and 4 are granted terminating mode and will act as the ENTER key (the code generated by the key will however appear at the end of the line).
- line 40: Input of data is requested from the link terminal and the last character isolated. If the line is empty (ENTER key was the only one pressed) the program ends in line 900.
- line 100: The last character is compared against codes generated by special function keys 2, 3 and 4. If the code does not match an error message is printed at the user terminal and new data requested. If the code matches control is passed to a specific section of code where the rest of the line is processed.

A complete table of correspondence between special function keys and codes follows:

Special function key number	Generated decimal code
2	16
3	17
4	18
5	19
6	20
7	21
8	22
9	23
10	24

- line 900: The program ends and all special function keys are returned to normal mode.

CONTROLLING THE DISPLAY UNIT

The following program demonstrates how to prevent data from appearing on the display unit during an input operation. It also shows how to print partial lines to the left of the display.

```

FILE-TERM,LT
10 FILES TERM
20 DIM A$(40)
30 PRINT #1;CTL(74)
40 REM DISPLAY HAS BEEN ECHOED-OFF
50 LINPUT #1;A$
60 REM PROCESS CONFIDENTIAL DATA
...
90 REM PROGRAM RESPONSE IS IN VARIABLE Z
100 PRINT #1;CTL(73)
110 REM DATA WILL NOW BE PRINTED ON THE DISPLAY
120 PRINT #1;Z;" _"
130 LINPUT #1;A$
...
200 END

```

Programming the Link Terminal

- Comments: The ASCII file TERM is equated with a link terminal device.
- line 10: The link terminal is reset and the display unit is by default in the echo-on mode. All characters printed to it or input through the keyboard will get printed from left to right.
- line 30: Some confidential data is requested by the program. No character will be printed on the display as it has been echoed-off.
- line 60: The confidential data contained in string A\$ is processed and a numeric response prepared in variable Z.
- line 100: The display is returned to normal echo-on mode.
- line 110: Some non-confidential data needs to be entered by the user. The response computed by the program is printed to the left of the display and separated by 2 spaces from the user response which will appear on the same line. This effect is achieved by printing a line terminated by _ (underscore character).
- line 200: The link terminal is reset as the program ends and the display is cleared.

CONTROLLING MULTIPLE LINK TERMINALS

Since each link terminal is an ASCII file a BASIC program can simultaneously access up to 16 of these devices. In practice however one or more files will also be needed for data logging or retrieving, so a more reasonable limit of the number of link terminals controlled by a single program is actually 10.

The main difficulty to overcome is to prevent the program from waiting for data from one link terminal while another terminal has its input line ready. Instead the program should poll the next available link terminal device. This effect can be achieved by defining an error trapping routine to which errors returned by the system will be directed. Error number 300 is returned when an input operation is still in progress and no data is ready at a link terminal. The error trapping section should act as a dispatcher for link terminals when error 300 is returned.

A typical application will be structured as a sequence of input and output phases. By numbering the data input states 1,2 . . . , the program keeps track of the state of each link terminal. A one dimensional array could be used for that purpose. Two dimensional arrays should be used for keeping link terminal dependent data.

The following example logs data collected at five link terminals to a Basic formatted file. The program has two input phases which are numbered 1 and 2. When a link terminal is no longer needed its state is set to 0.

```
FILE-SOURC1,LT10
FILE-SOURC2,LT11
FILE-SOURC3,LT12
FILE-SOURC4,LT13
FILE-SOURC5,LT14
CREATE-LOGFL,100
```

```

10 FILES SOURC1,SOURC2,SOURC3,SOURC4,SOURC5
15 FILES LOGFL
20 DIM A$(40)
30 DIM S(5)
40 IF ERROR THEN 200
45 I=1
50 REM INPUT STATE #1 STARTS HERE
55 MAT S=CON
60 LINPUT #I;A$
65 REM DATA FROM LINK TERMINAL #I IS LOGGED TO LOGFL
70 PRINT #6;I,A$
80 PRINT #I;" ",CTL(71),"1"
85 PRINT #I;CTL(72),"2,3"
90 REM INPUT STATE #2 STARTS HERE
95 S(I)=2
100 LINPUT #I;A$
105 PRINT #I;CTL(71);"-1"
110 PRINT #I;CTL(72);"-2,-3"
115 IF A$='16 THEN 300
120 IF A$='17 THEN 140
130 GOTO 80
135 REM LINK TERMINAL RESUMES AT INPUT STATE 1
140 S(I)=1
150 GOTO 60
190 REM DISPATCHER FOR LINK TERMINALS STARTS HERE
200 E=SYS(0)
210 IF E#300 THEN 300
215 REM DETERMINE NEXT LINK TERMINAL TO POLL
220 IF I#5 THEN 240
230 I=0
240 I=I+1
250 REM PASS CONTROL ACCORDING TO LINK TERMINAL STATE
260 IF S(I)=0 THEN 220
270 IF S(I)=1 THEN 60
280 IF S(2)=2 THEN 100
290 REM LINK TERMINAL IS DEALLOCATED
300 ASSIGN *,I,R
310 S(I)=0
320 IF S(1)+S(2)+S(3)+S(4)+S(5)#0 THEN 220
330 END

```

Comments: Five link terminals are equated with ASCII files. A BASIC formatted file is created for logging data collected at the link terminals.

line 10: All files accessed by the program are allocated.

line 20: A unique 40 character long string is defined for holding inputs from the link terminals. A five element array is defined for recording each link terminal state.

line 40: The "error trapping" routine is defined at statement 200.

line 45: The I variable is initialized. It will be associated with the link terminal currently under operation and its values will range between 1 and 5.

line 55: The state array is initialized and each link terminal is considered to be in state 1.

Programming the Link Terminal

- line 60: Data is requested from the current link terminal. If no data has been entered yet control will transfer to "error trapping" section.
- line 70: Data has been received and will be logged to the LOGFL file along with the file number.
- line 80: The next phase is prepared:
- display unit is cleared
 - light number 1 goes on and is supposed to be labelled "LAST ITEM ?"
 - special function keys 2 (standing for a YES) and 3 (standing for a NO) are granted terminating mode.
- line 90: The state of the current link terminal is set to 2.
- line 100: A YES/NO answer is expected. If the response is not ready yet control transfers to "error trapping" section. Otherwise the answer is analyzed:
- a wrong answer results in restarting state 2
 - a YES answer results in the current link terminal becoming deallocated
 - a NO answer results in resuming program at state 1.
- line 200: The "error trapping" acts as follows:
- the error number is retrieved. If not a 300 type the link terminal is freed
 - the I variable is updated to reference the next available link terminal
 - control transfers to section of code corresponding to the state of the link terminal under operation.
- line 300: An unexpected error has happened or the link terminal has transmitted its LAST-ITEM. Its state is set to 0 and it becomes available to other programs.

INTERFACING WITH DEVICES ON AN HPIB

The Hewlett-Packard Interface Bus (referred to as HPIB in this discussion) has become an international standard for interfacing electronic instruments. The HP 3070A link terminal should be viewed as an integrated set of three HPIB devices as illustrated in figure G-1. The link terminal's bus to which up to thirteen additional HPIB devices may be connected can be remotely controlled by a program (if configured with the I option). The symbolic level of HPIB messages made available to the programmer is described in table G-1. The general syntax of an HPIB statement is:

```
PRINT #I;CTL(70),A$
```

where A\$ stands for the symbolic message.

Note: Refer to IEEE-488-1975 document for a complete description of the HPIB standard.

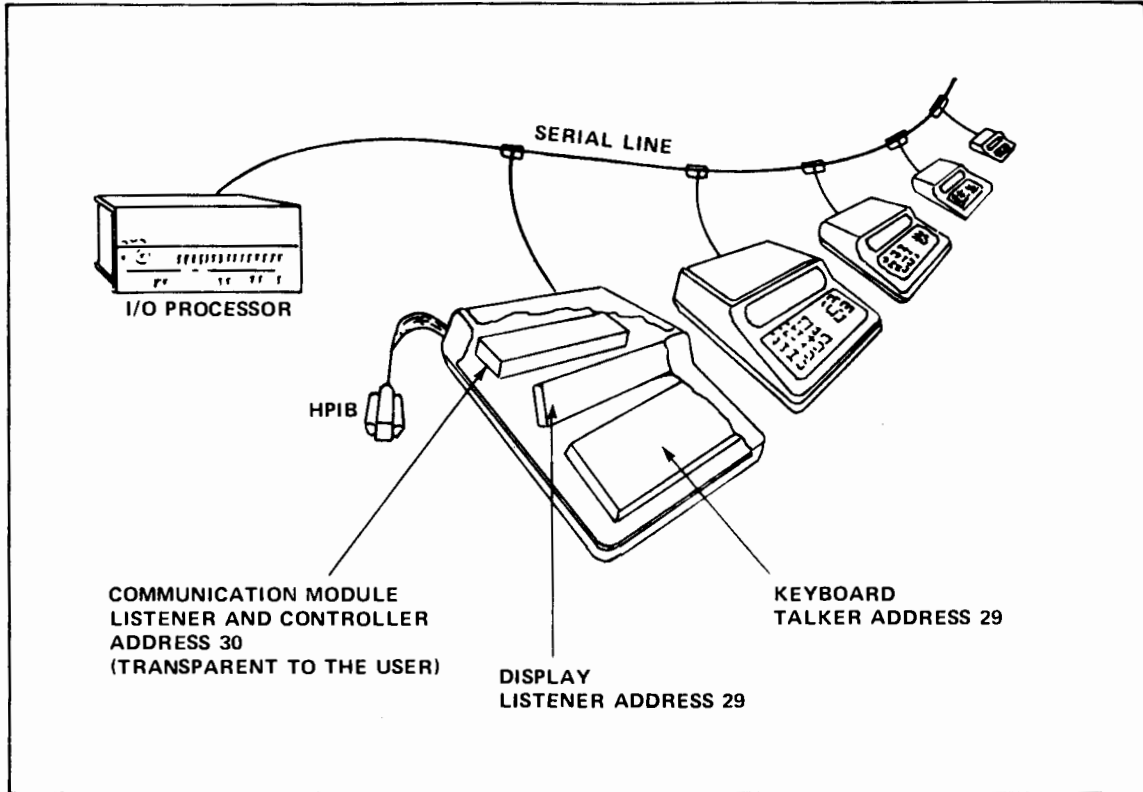


Figure G-1. The HP-IB Structure of the Link Terminal

Table G-1. Repertoire of HPIB Messages

FORMAT	MEANING
LSN-<d>	Configure device at listen address <d> as a listener.
TLK-<d>	Configure device at talk address <d> as talker (the actual configuration takes place when program requests an input).
UNL	Deconfigure all current listeners.
UNT	Deconfigure the current talker.
GET	Send Group-Execute-Trigger command to all current listeners.
DCL	Send Device-Clear command to responding devices.
SDC	Send Selected-Device-Clear to all current listeners.
REN	Send Remote-Enable command to responding devices.
GTL	Send Go-To-Local command to all current listeners.
LLO	Send Local-Lockout command to responding devices.
LEN	Send Local-Enable command to responding devices.
STB-<d>	Request status byte from device at talk address <d>. The status may be read through an input statement (READ or LINPUT) and is returned as two integers. The first reflects the state of the SRQ line (value is 0 or 1), the second is the actual device dependent status.
SQI-<d>	Identification of the Service Requested condition from device at address <d>. The information returned is either <d> if that device requested service or 31 otherwise. It is obtained through an input statement.
TCT-<d>	Pass control to an active controller at address <d>. The controller should treat the program as a device at listen and talk address 0. Control is passed back by receiving a TCT-30 from the controller.
IFC	Abort operations at responding devices and return them to their initial state.
NOTES: 1. Spaces, "-" and "," are acceptable separators between a command and the argument. 2. The <d> argument should be a positive number between 1 and 29.	

The following programs illustrate the use of the HPIB. The first program is self documented and controls digital to analog data conversion with local results printout. The second program shows how an HP 9825A calculator can be delegated control of the bus.

Example 1: Digital to analog data conversion.

```

FILE-TERM,LT
10  FILES TERM
15  DIM A$(40),V$(20)
20  PRINT "THE FOLLOWING HPIB INSTRUMENTS ARE USED IN"
25  PRINT "THIS EXAMPLE : 59303A D/A CONVERTER"
30  PRINT "                : 5312A ASCII INTERFACE"
35  PRINT "                : 5306A MULTIMETER/COUNTER"
40  PRINT "                : 5150A COLUMN PRINTER "
50  PRINT
55  PRINT "SPECIFY CONFIGURATION OF THE HPIB "
60  PRINT "ADDRESS OF 59303A ";
65  INPUT S1
70  PRINT "ADDRESS OF 5312A  ";
75  INPUT S2
80  PRINT "ADDRESS OF 5150A  ";
85  INPUT S3
90  REM  SELECT OUTPUT AND CONVERSION MODE FOR 59303A
95  PRINT #1;CTL(70),"REN"
100 PRINT #1;CTL(70),"LSN",S1+1
105 PRINT #1;CTL(70),"LLO"
110 PRINT #1;"E0"
115 REM  OBTAIN VOLTAGE TO CONVERT
120 PRINT "ENTER VOLTAGE OR RETURN TO END ? ";
125 LINPUT V$
130 IF LEN(V$)=0 THEN 230
135 REM  SEND VOLTAGE TO 5312A INTERFACE
140 PRINT #1;CTL(70),"UNL"
145 PRINT #1;CTL(70),"LSN",S1
150 PRINT #1;V$
155 REM  SEND INITIATE COMMAND TO 5312A INTERFACE
160 PRINT #1;CTL(70);"UNL"
165 PRINT #1;CTL(70);"LSN",S2
170 PRINT #1;"I"
175 REM  PREPARE 5306A TO SEND MEASURED VOLTAGE
180 PRINT #1;CTL(70);"UNL"
185 PRINT #1;CTL(70);"TLK",S2
190 REM  PRINT VOLTAGE ON DISPLAY , 5150A AND TERMINAL
195 PRINT #1;CTL(70);"LSN-29"
200 PRINT #1;CTL(70),"LSN",S3
205 LINPUT #1;A$
210 PRINT "VOLTAGE READ FROM 5306A : ";A$
225 GOTO 115
230 END

```

Example 2: Delegation of control to a local controller.

In order to achieve greater speed on the HPIB and unload the System, a program can delegate control of the HPIB to a local controller. In the following example an HP 9825A calculator plays the role of the intelligent controller. Once it has obtained control of the bus it treats the program as an HPIB device with talk and listen address 0. The program exchanges data with the other devices on the bus under the supervision of the active controller, but also remains the system controller with address 30. Control is passed back to the program by sending it the proper HPIB command.

Note: Refer to the HP 9825A Extended I/O manual before reading this program.

Basic program

HPL program

FILE-HPIB,LTS

<pre> 10 FILES HPIB 20 I=1 <---Request Service--- 30 PRINT #1;CTL(70),"STB-21" 40 READ #1;Q,S 50 IF Q=0 THEN 30 60 PRINT #1;CTL(70),"LSN-21" 70 PRINT #1;I ---Send Information---> 80 PRINT #1;CTL(70);"TCT-21" ---Delegate Control---> 90 ... <---Send Result----- 200 READ #1;R <---Pass Control----- 210 IF R=0 THEN 300 220 I=I+1 230 GOTO 30 300 END </pre>	<pre> 0 : dev "PROG",700,"SYSTEM",730 1 : oni 7, "INTER" 2 : eir 7,80 3 : rqs 7,0 4 : gto 4 5 : "INTER" : rds(7)->S 6 : if bit(4,S)=1;gsb"INFO" 7 : if bit(6,S)=1;gsb"CNTRL" 8 : iret 9 : "INFO" : red "PROG",P;ret 10 : "CNTRL" : 20 : wrt "PROG",R 21 : pct "SYSTEM" 22 : ret 23 : end </pre>
--	---

Comments: The ASCII file HPIB is equated with a link terminal. An HP 9825A calculator is connected to it through the HPIB and is assumed to have the proper ROM. It also has to be configured as a non system controller.

line 10: The link terminal is initialized and an IFC is sent to the bus to abort current activity.

line 30: The status byte information is requested from the calculator which causes a serial poll on device 21 (the calculator) to occur.

line 40: The SRQ bit and status information are read. The calculator is interrogated until it requests service.

line 60: The calculator is configured as a listener and data is sent to it.

line 80: Control is delegated to the calculator. While it now manages activity of devices on the bus the program can perform an independent task until it needs to exchange data with the bus.

line 200: The program reads a numeric result and obtains control back.

line 300: The program aborts all operations on the bus as it clears the link terminal with an IFC.

A

ABS function, definition, 11-20
 Access restriction codes, 8-6, 8-8
 Access restriction codes, list, 10-12
 Access restrictions, files, 8-8
 Access restrictions, programs, 8-6
 2000 to 2000 communication, 1-3, 9-1, 9-20

 Account access, 8-5
 Account number, 1-3
 Account structure, overview, 1-6
 ADVANCE statement, definition, 11-20
 ADVANCE statement, usage, 5-10
 APPEND command, definition, 10-11
 APPEND command, usage, 2-30
 Arithmetic operators, 2-5, 11-11
 ASCII character set, A-1
 ASCII character set, usage, 4-1
 Array, definition, 11-1
 Array, usage, 3-1
 Array dimensioning, 3-2, 11-39
 Array element, definition, 11-2
 Array element, usage, 3-1
 Array name, definition, 11-2
 Array name, usage, 3-1
 Array reference, 3-1, 11-2
 Array size, logical, 11-7
 Array size, physical, 11-2
 Arrays, adding or subtracting, 3-9, 11-58
 Arrays, assigning values to, 3-3, 11-59, 11-66
 Arrays, conversion of, 3-8, 11-59
 Arrays, formatted output, 11-65
 Arrays, initialization of, 3-7, 11-59, 11-68
 Arrays, inversion of, 3-14, 11-60
 Arrays, multiplication of, 3-11, 11-61
 Arrays, new dimensions, 3-7, 11-8
 Arrays, printing of, 3-4, 11-62
 Arrays, scalar multiplication of, 3-9, 3-16, 11-67
 Arrays, transposition of, 3-15, 11-67
 ASCII disc file, CTL functions, 5-22, 11-32
 ASCII file, definition, 10-2
 ASCII file, usage, 5-1, 5-17
 ASCII file errors, C-13
 ASCII files, access restrictions, 8-8
 ASCII files, accessing 5-20, 5-23, 11-76, 11-87
 ASCII files, closing, 5-20, 11-21
 ASCII files, creation, 5-17, 10-16
 ASCII files, opening, 5-20, 11-41
 ASCII files, print to, 5-20, 11-76
 ASCII files, purging, 5-17, 10-25, 11-83
 ASCII files, read from, 5-23, 11-87
 ASP remote command summary, 9-17
 ASSIGN statement, definition, 11-20a

ASSIGN statement, usage, 5-5, 5-20
 Assignment statement, definition, 11-54
 Assignment statement, usage, 2-9
 ATN function, definition, 11-23
 AUTO LF key, 1-9

B

Baud rates, 1-9
 BASIC commands, definitions, 10-10
 BASIC commands, use of, 2-26
 BASIC formatted file, definition, 10-5
 BASIC formatted file, usage, 5-1, 5-2
 BASIC formatted files, accessing, 5-6
 BASIC formatted files, serial access, 5-6, 11-74, 11-85, 11-86
 BASIC formatted files, direct access, 5-12, 11-75, 11-87
 BASIC formatted files, closing, 5-5, 11-20
 BASIC formatted files, creation, 5-3, 10-13, 11-30
 BASIC formatted files, opening, 5-5, 11-41
 BASIC formatted files, print to, 5-7, 5-12, 11-74
 BASIC format files, purging, 5-5, 10-25, 11-83
 BASIC formatted files, read from, 5-9, 5-14, 11-85
 BASIC functions, 11-19
 BASIC language elements, 2-1
 BASIC language terms, 11-1
 BASIC statements, definitions, 11-19
 BASIC statements, usage, 2-9
 BASIC subroutines, usage, 2-22
 BESTOW, operator command, E-1
 Block, definition, 10-5
 BNF syntax, F-1
 Boolean constants (see logical constants)
 Boolean operators (see logical operators)
 Break capability, 11-23
 BREAK key, programmatic detection of, 7-5
 BRK function, definition, 11-23
 BRK function, usage, 7-5
 BYE command, definition, 10-11
 BYE command, usage, 1-12

C

Card reader as ASCII file, 10-3
 Card reader, CTL functions, 11-33, 11-35
 Carriage control characters, 6-5, 11-77
 Cartridge use, HP 2644, 2-35
 CATALOG command, definition, 10-12
 CDC, communication with 9-3
 CDC EXPORT/IMPORT commands 9-18
 CDC manuals for use with RJE, 9-19
 CHAIN statement, definition, 11-25
 CHAIN statement, usage, 7-1
 Character, definition, 11-3

INDEX (Continued)

Character set, ASCII, A-1, A-2
Character set, EBCDIC, A-3, A-4
CHR\$ function, definition, 11-27
CHR\$ function, usage, 4-7, 4-8
COM statement, definition, 11-28
COM statement, usage, 7-3
Comma in PRINT statement, 2-10
Commands, definitions, 10-10
Commands, use of, 2-26
Commands, programmatic execution of, 7-6
Comments (see Remarks)
Common data, 7-3
Comparing strings, 4-7
Comparison of values, 11-48
Compilation errors, C-6
CON function (see MAT CON)
Connecting to system, 1-7
Constant, definition, 11-3
Constant, string (see literal string)
Controlling Multiple Devices, 9-15
CONVERT statement, definition, 11-29
CONVERT statement, usage, 4-7
COPY, operator command, E-1
COS function, definition, 11-29
CREATE statement, definition, 11-30
CREATE statement, usage, 5-3
CREATE command, definition, 10-13
CREATE command, usage, 5-3
CSAVE command, definition, 10-27
CTL function, definition, 11-31
CTL function, usage, 5-22

D

Data communications, 9-1
DATA statement, definition, 11-37
DATA statement, usage, 2-17
Data transmission, 2000 to 2000, 9-20
Data transmission between systems, 9-21
Data transmission between programs, 9-22
Dedicated application environment, 1-7
Dedicated application, usage, 8-4
DEF statement, definition, 11-38
DEF statement, usage, 2-25
DELETE command, definition, 10-13
DELETE command, usage, 2-28
Delimiters, formatted print, 6-5, 11-78
Destination string, definition, 11-3
DEVICE command, definition, 10-14
DEVICE command, usage, 5-17, 5-18
Device designator, definition, 10-5
Device designator, usage, 5-18, 5-19
Device, non-sharable, 10-8
DIM statement, definition, 11-39
DIM statement, use with arrays, 3-2

DIM statement, use with strings, 4-3
Dimensioning arrays, 3-2
Dimensioning strings, 4-3
Dimensions, new in array, 3-6, 11-8
Direct file access, 5-12, 11-75, 11-87
Direct file read, 5-14, 11-85
DUMP, operator command, E-2
DUPLEX switch, 1-8, 10-6, 10-7
Dynamic access to files, 11-21

E

EBCDIC character set, A-3
ECHO command, definition, 10-15
ECHO command, usage, 1-11
ECHO/NO ECHO switch, 1-8
Element, array, 3-1, 11-2
Emulator, 2770/2780/3780, 9-1, 9-4, 9-23
END statement, definition, 11-39
END statement, usage, 2-14
End-of-file mark, definition, 10-6
End-of-file mark, usage, 5-3, 5-5, 5-8
End-of-file operations, 11-49
End-of-record mark, 5-5, 5-6
ENTER statement, definition, 11-40
ENTER statement, usage, 2-21
ENTER statement, use with strings, 4-5
EOF (see end-of-file mark)
EOR (see end-of-record mark)
Error correction, 1-10
Error detection, programmatic, 7-4
Error messages, by command, C-1
Error messages, syntax errors, C-5
Error messages, execution errors, C-6
Errors, formatted print, 11-81
Errors, trapping in program, 7-4
EXECUTE command, definition, 10-15
EXECUTE command, usage, 2-31
Executing programs, 2-29, 2-31
Execution errors, C-6
EXP function, definition, 11-41
Expression, logical, 2-6, 11-1
Expression, numeric, 2-5, 11-9
Expression, relational, 2-5, 11-11
Expression, string, 11-16
Expressions, evaluation of, 2-6
Expressions, use of 2-4

F

False value, 11-48
FCP (see File Create/Purge)
File access, BASIC formatted files, 5-6
File access, ASCII files, 5-20

INDEX (Continued)

File access, ASCII files, 5-20
File access restrictions, dynamic, 11-21
File access restrictions, static, 11-22
File access restrictions, usage, 8-7
FILE command, definition, 10-16
FILE command, usage, 5-17
FILE command, use with job functions designators, 9-8
File Create/Purge capability, 8-5
File descriptors, list, 10-12
File length, definition, 10-6
File length, usage, 5-3, 5-19
File name, definition, 10-6, 11-5
File number, definition, 11-5
File space, 1-3
File states, 8-7
Files, close ASCII, 5-20, 11-20
Files, close BASIC formatted, 5-5, 11-20
Files, creation of ASCII, 5-17, 10-16
Files, creation of BASIC formatted, 5-2, 10-13, 11-30
Files, difference between ASCII and BASIC formatted, 5-19
Files, formatted print to, 11-82
Files, open ASCII, 5-20, 11-41
Files, open BASIC formatted, 5-5, 11-41
Files, print arrays to, 11-64
Files, print formatted arrays to, 11-65
Files, print to ASCII, 5-20, 11-76
Files, print to BASIC formatted, 5-6, 5-13, 11-74
Files, purge ASCII, 5-19, 10-25
Files, purge BASIC formatted, 5-5, 10-25
Files, read arrays from, 11-66
Files, read from ASCII, 11-87
Files, read from BASIC formatted, 5-9, 5-14, 11-85
FILES statement, definition, 11-41
FILES statement, usage, 5-5, 5-20
Files, update, 5-9
Files, usage, 5-1
Fixed-point constants, 2-2
Fixed-point formatted print format, 11-79
Fixed-point print format, 11-73
Floating-point constants, 2-2
Floating-point formatted print format, 11-80
Floating-point print format, 11-73
FOR loops, nesting of 11-44
FOR statement, definition, 11-43
FOR statement, usage, 2-15
FOR variable, 2-16, 11-43
Format characters, formatted print, 11-77
Format errors, C-11
Format specification, formatted print, 11-78
Format string, formatted output, 6-3
Formatted output, definition, 11-77
Formatted output, usage, 6-1, 6-7
Formatted output, arrays, 11-65
Formatted print, 6-1, 11-77
Forms control, RJE output, 9-10
Full duplex, definition, 10-6

Full duplex, usage, 1-8
FULL DUPLEX/HALF DUPLEX switch, 1-8
Function, definition, 11-5
Function, usage, 2-8
Function reference, definition, 11-6
Functions, defining your own, 2-25
Functions, list, 11-5

G

General device designators, 10-5, 10-6
GET command, definition, 10-17
GET command, usage, 2-30
GROUP command, definition, 10-12
GROUP command, usage, 2-31
Group library, definition, 10-7
Group library, usage, 8-3
Group master, 8-3
Group master, accounts of, 1-6, 8-2
Groups, formatted print, 11-78
GOSUB statement, definition, 11-45
GOSUB statement, usage, 2-22
GOTO statement, definition, 11-47
GOTO statement, usage, 2-13

H

Half duplex, definition, 10-7
HASP remote command summary, 9-17
HASP workstation, 9-3
HELLO command, definition, 10-18
HELLO command, usage, 1-9
HELLO program, 10-19
HELLO program, usage, 1-7, 8-4
Host system manuals, 9-19
Host systems for RJE, 9-3
Host functions, 9-3
Host function/job function communication, 9-5, 9-20
HP 2640/2644 terminal interface, D-3
HP 2644, using cartridges with, 2-35

I

IBM, communication with, 9-3
IBM manuals, for RJE use, 9-19
IBM remote commands, for RJE use, 9-17
IBM 2741 terminal interface, D-6
Icode, definition, 10-7
Icode, usage, 1-9, 8-1
IDN function (see MAT IDN)
IF statement, definition, 11-48
IF statement, usage, 2-14

INDEX (Continued)

IF END statement, definition, 11-49
IF END statement, usage, 5-8
IF ERROR statement, definition, 11-50
IF ERROR statement, usage, 7-4
IMAGE statement, definition, 11-50
IMAGE statement, usage, 6-2
Initializing arrays, 3-7
Integer constants, 2-1
Integer print format, 11-72
Integer format, formatted print, 11-79
Input from terminal, 2-20, 2-21
INPUT statement, definition, 11-51
INPUT statement, usage, 2-20
INPUT statement, use with strings, 4-5
Input-output devices, 1-3
INT function, definition, 11-52
INV function (see MAT INV)
ITM function, definition, 11-53
ITM function, usage, 5-15

J

Job function designator, definition, 10-8
Job function designator, usage, 9-4
Job function designator, use in FILE command, 9-8
Job function designators, list, 10-6
Job function/host function communication, 9-5, 9-20
Job inquiry file, usage, 9-13, 9-14
Job list file, usage, 9-9, 9-13, 9-21
Job message file, usage, 9-14
Job transmitter file, usage, 9-8, 9-13, 9-21

K

KEY command, definition, 10-19
KEY command, usage, 2-34

L

LEN function, definition, 11-53
LEN function, usage, 4-9, 4-10
LENGTH command, definition, 10-19
LENGTH command, usage, 2-29
LET statement, definition, 11-54
LET statement, usage, 2-9
LET statement, use with strings, 4-5
Letters, upper/lower case, 4-2
Library, definition, 10-8
Library, usage, 2-30, 8-3
Library catalog, 2-31
LIBRARY command, definition, 10-12
LIBRARY command, usage, 2-31
Library, operator commands, E-1
LIN function, definition, 11-55

Link terminal, messages, G-8
Link terminal, ASCII file, 10-4
Link terminal, CTL functions, 11-33, 11-37
LIN function, usage, 2-12
Line printer, ASCII file, 10-2
Line printer CTL functions, 11-32, 11-34
LINE/LOCAL switch, 1-8
Linking programs, 7-1
LINPUT statement, definition, 11-56
LINPUT statement, usage, 4-5
LINPUT≠ statement, definition, 11-56
LINPUT≠ statement, usage, 5-25
LIST command, definition, 10-20
LIST command, usage, 2-27
Literal format, formatted print, 11-80
Literal string, definition, 11-7
Literal string, usage, 2-2, 4-1
Literal string, in formatted output, 6-6
LOAD command, definition, 10-20
LOAD, operator command, E-2
LOCK command, definition, 10-21
LOCK command, usage, 8-6, 8-7
LOCK statement, definition, 11-57
LOCK statement, usage, 8-8, 8-9
Locked files, 8-7, 8-8
Locked programs, 8-6, 8-7
LOG function, definition, 11-58
Logging on, 1-9
Logical constants, 2-3
Logical expression, 11-11
Logical length, 11-7
Logical operators, 2-6
Logical size, definition, 11-7
Looping statements, 2-15, 11-43
Lower-case letters, 4-2

M

Magnetic tape, ASCII file, 10-4
Magnetic tape cartridge, use of, 2-35
Magnetic tape CTL functions, 11-32, 11-34
MASK, 11-21
MAT addition statement, definition, 11-58
MAT addition statement, usage, 3-9, 3-10
MAT assignment statement, definition, 11-59
MAT assignment statement, usage, 3-3
MAT CON statement, definition, 11-59
MAT CON statement, usage, 3-7, 3-8
MAT IDN statement, definition, 11-59
MAT IDN statement, usage, 3-7, 3-8
MAT INPUT statement, definition, 11-60
MAT INPUT statement, usage, 3-4
MAT INV statement, definition, 11-60
MAT INV statement, usage, 3-9, 3-14
MAT multiplication statement, definition, 11-61
MAT multiplication statement, usage, 3-9, 3-11

MAT print statement, definition, 11-62
MAT PRINT statement, usage, 3-4
MAT PRINT# statement, definition, 11-64
MAT PRINT USING statement, definition, 11-65
MAT PRINT USING statement, usage, 6-1
MAT PRINT# USING statement, definition, 11-65
MAT PRINT# USING statement, usage, 5-21
MAT READ statement, definition, 11-66
MAT READ statement, usage, 3-3
MAT READ# statement, definition, 11-66
MAT scalar multiplication statement, definition, 11-67
MAT scalar multiplication statement, usage, 3-16
MAT subtraction statement, definition, 11-58
MAT subtraction statement, usage, 3-9, 3-10
MAT TRN statement, definition, 11-67
MAT TRN statement, usage, 3-9, 3-15
MAT ZER statement, definition, 11-68
MAT ZER statement, usage, 3-7
MAX operator, 2-5, 11-9
MESSAGE command, definition, 10-21
MESSAGE command, usage, 1-12
MIN operator, 2-5, 11-9
MIN/MAX, 11-9
Multi-branch GOSUB statement, definition, 11-46
Multi-branch GOSUB statement, usage, 2-24
Multi-branch GOTO statement, definition, 11-47
Multi-branch GOTO statement, usage, 2-13
Multiple-Write-Access, 8-5
MWA command, definition, 10-22
MWA command, usage, 8-8

N

NAME command, definition, 10-22
NAME command, usage, 2-29
Nested FOR loops, definition, 11-44
Nested FOR loops, usage, 2-16
Nested GOSUB statements, definition, 11-46
Nested GOSUB statements, usage, 2-24
New dimensions, definition, 11-8
NEXT statement, definition, 11-43
NEXT statement, usage, 2-15
NUM function, definition, 11-69
NUM function, usage, 4-10
Number, definition, 11-8
NOFCP (see File Create/Purge)
Non-sharable device, definition, 10-8
Non-sharable devices, list, 10-5
NOPFA (see Program/File Access)
Numeric constant, definition, 11-8
Numeric constant, usage, 2-1
Numeric equivalent of character, 4-2
Numeric expression, definition, 11-9
Numeric expression, usage, 2-4

Numeric formatting characters, 6-3
Numeric output formats, 11-72
Numeric simple variable, definition, 11-12
Numeric to string conversion, 11-29
Numeric valued functions, list, 11-5
Numeric variable, definition, 11-12
Numeric variable, usage, 2-3

O

ON-LINE mode, 1-7
Operators, 2-5
Operators, hierarchy of, 11-10
Operators, relational, 11-13
OUT=filename, definition, 10-8
OUT=filename, usage, 5-23
Output, formatting, 6-1

P

Paper tape, CTL functions, 11-32, 11-33, 11-34
Paper tape, preparing off-line, B-1
Paper tape, printing to, 2-33
Paper tape, reading from, 2-34
Paper tape punch, ASCII file, 10-3
Paper tape reader, ASCII file, 10-2
Parameters, passing between programs, 7-3
Password, 1-3, 1-9
PAUSE command, definition, 10-23
PAUSE command, usage, 7-7
PFA (see Program/File Access)
Physical length, definition, 11-12
Physical size, definition, 11-12
POS function, definition, 11-69
POS function, usage, 4-11
Positioning file, 5-10
Primary, definition, 11-13
Print, serial file, 5-6
Print, direct file, 5-13
Print formats, numeric, 11-72
Print functions, definition, 11-70
Print functions, usage, 2-11
PRINT statement, definition, 11-70
PRINT statement, usage, 2-10
PRINT statement, use with strings, 4-5, 4-11
PRINT# statement, definition, 11-74
PRINT# statement, usage, 5-16, 5-13, 5-20
PRINT USING statement, definition, 11-77
PRINT USING statement, usage, 6-1
PRINT USING statement, use with strings, 4-12
PRINT# USING statement, definition, 11-82
PRINT# USING statement, usage, 5-21
PRIVATE command, definition, 10-23

INDEX (Continued)

PRIVATE command, usage, 8-6
Private files 8-7, 8-8
Private library, 8-3
Private programs, 8-6
Program access, 11-25
Program descriptors, list, 10-12
Program designator, definition, 11-13
Program execution, 2-29
Program/File Access, 8-5
Program name, definition, 10-9, 11-13
Program reference, definition, 10-9
Program states, 8-6
Programming errors, C-7
PROTECT command, definition, 10-24
PROTECT command, usage, 8-6, 8-7
Protected files, 8-7, 8-8
Protected programs, 8-6, 8-7
PUNCH command, definition, 10-24
PUNCH command, usage, 2-33, 2-35
PURGE command, definition, 10-25
PURGE command, use with files, 5-5, 5-19
PURGE command, use with programs, 2-31
PURGE statement, definition, 11-83
PURGE statement, usage, 5-5

Q

Qualifying program name, 10-9

R

Read, direct file, 5-14
Read, serial file, 5-9
READ statement, definition, 11-84
READ statement, usage, 2-17
READ statement, use with strings, 4-6
READ= statement, definition, 11-85
READ= statement, usage, 5-9, 5-14, 5-23
Reader/punch/interpreter, ASCII file, 10-3
Reader/punch/interpreter CTL functions, 11-33, 11-35
REC function, definition, 11-88
REC function, usage, 5-15
Record, definition, 10-9
Record, direct reference, 5-12
Record length, definition, 10-9
Record number, definition, 11-13
Redimensioning arrays, 3-6
Referencing arrays, 3-1
Referencing strings, 4-3
Relational operator, definition, 11-13
Relational operator, usage, 2-5
Relational operator, use with strings, 4-7
REM statement, definition, 11-88
REM statement, usage, 2-13
Remarks, definition, 11-88
Remarks, use in program, 2-13
Remote job entry, 9-1
Remote job entry, via card reader, 9-6

Remote job entry, programmatic, 9-7
Remote job entry, retrieving output, 9-9, 9-10
REMOTE LOCAL switch, 1-8
RENUMBER command, definition, 10-26
RENUMBER command, usage, 2-27
Repetition factor, formatted print, 11-78
RESTORE statement, definition, 11-89
RESTORE statement, usage, 2-17
RESTORE, operator command, E-2
RETURN statement, definition, 11-45
RETURN statement, usage, 2-22
Return variable, definition, 11-14
RJE, CTL functions, 11-36
RJE, definition, 1-3
RJE, usage, 9-1
RJE commands, 9-11
RND function, definition, 11-90
RUN command, definition, 10-27
RUN command, usage, 2-29

S

SAVE command, definition, 10-27
SAVE command, usage, 2-29
Scalar multiplication, arrays, 3-16
SCRATCH command, definition, 10-28
SCRATCH command, usage, 2-30
Security, 1-3
Semicolon, use with PRINT, 2-10
Serial file access, 5-6
Serial file read, 11-85
SGN function, definition, 11-91
SIN function, definition, 11-91
Single-Write Access, usage, 8-5, 8-8
SPA function, definition, 11-92
SPA function, usage, 2-12
Specific device designators, 10-5
SQR function, definition, 11-92
Source string, definition, 11-14
Statement number, definition, 10-9, 11-15
Statement number, usage, 2-9
Statements, list, 11-19
Statements, usage, 2-9
STEP size, 2-16, 11-43
STOP statement, definition, 11-92
STOP statement, usage, 2-14
String, definition, 11-16
String, usage, 4-1
String, source, 11-14
String, destination, 11-3
String character set, 4-1
String comparison, 4-7
String constant (see literal string)
String data, assignment of, 4-5
String expression, definition, 11-16
String expression, usage, 2-4



String format, formatted print, 11-80
String formatting characters, 6-3
String length, definition, 11-16
String length, logical, 11-7
String length, physical, 11-12
String/numeric conversion, 11-29
String simple variable, definition, 11-17
String-valued functions, list, 11-6
String-valued functions, usage, 4-6
String variable, definition, 11-17
String variable, usage, 2-3, 4-3
Subroutines, usage, 2-22
Subscripted variable, definition, 11-17
Subscripted variable, usage, 2-4
Subscripts, use with arrays, 3-1
Subscripts, use with strings, 4-4
Substring designator, definition, 11-18
Substrings, usage, 4-4
SWA (see Single-Write-Access)
SWA command, definition, 10-28
SWA command, usage, 8-8
Syntax, BNF, F-1
Syntax errors, C-5
SYS function, definition, 11-93
SYS function, usage, 7-4, 7-5
System capabilities, 1-4
System hardware, 1-1
System library, definition, 10-10
System library, usage, 8-3
System manager, 1-1
System master, 8-3
System master account, 1-6, 8-2
System operator, 1-1
System operator commands, E-1
System operator functions, 1-12
SYSTEM statement, definition, 11-94
SYSTEM statement, usage, 7-6

T

TAB function, definition, 11-95
TAB function, usage, 2-12
TAN function, definition, 11-95
Tape Cartridge, 2-35, D-4
TAPE command, definition, 10-29
TAPE command, usage, 2-34, 2-36
Terminal type, detection of, 7-5
Terminal type, list, D-3
Terminal type, usage, 1-7, 1-10
Terminal speed, 1-9
Terminal time, 1-3
Terminals used with 2000 System, D-1
TIM function, definition, 11-96

TIME command, definition, 10-29
TIME command, usage, 1-12
Timed input, 2-21
Trapping errors, 7-4
TRN function (see MAT TRN)
True value, 11-48
TYP function, definition, 11-97
TYP function, usage, 5-14

U

UNLOCK statement, definition, 11-98
UNLOCK statement, usage, 8-8, 8-9
UNRESTRICT command, definition, 10-30
UNRESTRICT command, usage, 8-6, 8-7
Unrestricted files, 8-7, 8-8
Unrestricted programs, 8-6, 8-7
Update file, 5-9
UPDATE# statement, definition, 11-99
UPDATE# statement, usage, 5-10
Upper-case letters, 4-2
UPS\$ function, definition, 11-100
UPS\$ function, usage, 4-9
User accounts 1-6
User-defined functions, definition, 11-6
User-defined functions, usage, 2-25
USER 200 host system, 9-3
Using list, formatted output, 6-2

V

Variable, numeric, 11-12
Variable, string, 11-17
Variable, subscripted, 11-17
Variables, 2-3

W

Warning messages, C-12
Work space, definition, 10-10
Work space, usage, 2-26

Z

ZER function (see MAT ZER)
=, assignment operator, 2-10
OUT=filename, definition, 10-28
OUT=filename, usage, 5-23
2000 to 2000 Communications, 9-20
2770/2780/3780 Emulator, 9-1, 9-4, 9-23

