BASIC Programming

Part No. 09845-93000
Microfiche No. 09845-96000

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525

Copyright by Hewlett-Packard Company 1981

Printing History

This manual is for use with the HP 9845B/C. It is a revised version of the Operating and Programming manual, part number 09845-92000. Chapters 1 and 2 have been replaced by a shorter version of the Beginner's Guide, part number 09845-92001. (The Beginner's Guide will no longer be available.)

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the change on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

April 1981...First Edition. Updated pages: 34, 51, 58, 68, 75, 76, 78, 80, 82, 83, 85, 90, 96, 119, 127, 130, 131, 140, 166, 171, 173, 176, 178, 180, 183, 186, 204, 210, 217, 239, 240, 242, 246, 249

August 1981...Second Edition. Updated pages: ii, 1, 14, 191, RT-21

September 1981 ... Third Edition. Updated pages: BP-98, BP-171, BP-189, RT-4, EM-4, EM-5

09845-93000, rev: 9/81

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Preface

The BASIC Programming Manual is one of the manuals provided with your 9845 Computer. It is designed to be used by all 9845 users, from those who have never programmed to those who have programmed extensively using the BASIC language.

If you are a beginning programmer, you should read the Programming Tutorial in Chapter 2. You can also use the Introductory Training Tape and Workbook to become familiar with your computer.

When you are comfortable with BASIC and programming, you can find all of the 9845 mainframe statements discussed in detail in this manual. You can also use the pocket-sized Quick Reference as a handy guide to the language syntax.

In general, this manual groups the various BASIC statements, functions and commands together by topic. For example, all output statements and functions are found in Chapter 10. As much as possible, major topics are self-contained so you don't need to read an entire chapter to extract one idea. However, in some instances, statements that haven't been introduced are used to help illustrate the topic being discussed; the PRINT statement is used frequently this way. It is recommended that you read Chapter 1 of the Installation, Operation and Test Manual and Chapter 1 of this manual to get acquainted with the computer. After that, you need only read about the topics you want.

The coverage of each statement, function and command is restricted to its syntax, rules for its usage and some ways to use it (shown in text or in an example). The example programs are not intended to be comprehensive, but to illustrate syntax and a typical usage. In some cases, certain lines are emphasized by a bullet (●) placed next to them.

Table of Contents

Printing History ii

Preface iii

Chapter 1: General Information

The Keyboard 1
Alphanumeric and Numeric Keys 1
General Purpose Keys 1
Typing, Line Editing and CRT Display Keys 1
Special Function Keys 1
Resetting the Computer 2
Logging Keyboard Operations 2
The PRINT ALL IS Statement 2
Using the Keyboard While a Program is Running 3
The SUSPEND INTERACTIVE Statement 3
The RESUME INTERACTIVE Statement 4
Error Messages and Warnings 4
9845A vs. 9845B/C 4

Chapter 2: Programming Tutorial

Where Do You Go From Here? 5
Problem Solving: Defining the Problem 6
Problem Solving: Developing a Solution 6
Program Outlines 6
Flowcharts 7
Problem Solving: Writing the Program 8
More Examples 10

Chapter 3: Programming

Syntax Conventions 16
Programming Terms 16
Program Fundamentals 18
Entering Program Lines 19
The EDIT LINE Command 19
Increment Value 20
Automatic Indent 20
Inserting Lines 20
Deleting Lines 21
The DEL Command 21

Exiting the Edit Line Mode 22
The AUTO Command 22
The REN Command 22
Spacing 23
Space Dependent Mode 23
Remarks 25
The REM Statement 25
Comment Delimiter 25
The LIST Command 26
Available Memory 26
Alternate Printing Devices (LIST#) 26
The RUN Command and [RUN] 27
THE STEP Key 27
The PAUSE Statement and [PAUSE] 28
The CONTINUE Command and [CONT] 28
Terminating Execution 28
The STOP Key 28
The STOP Statement 29
The END Statement 29
Reset 29
The SCRATCH Command 30
The SECURE Statement 30
Miscellaneous Statements 31
The WAIT Statement 31
The TYPEWRITER ON Statement 31
The TYPEWRITER OFF Statement 31

Chapter 4: Mathematics

Operators 34
Arithmetic Operators 34
Relational Operators 34
Logical Operators 35
AND Operator 35
OR Operator 35
EXOR Operator 36
NOT Operator 36
DIV Operator 37
MOD Operator 37
Range 38
Number Formats 38
The STANDARD Statement 39
The FIXED Statement 39
The FLOAT Statement 40
Rounding 41
Significant Digits 41
Math Functions and Statements 42
General Functions 42
Logarithmic and Exponential Functions 46
Trigonometric Functions and Statements 46

Math Hierarchy 48
Parentheses 49
Math Errors-Recovery 50
The DEFAULT ON Statement 50
The DEFAULT OFF Statement 51

Chapter 5: Using Variables

Types 54
Forms 54
Names 55
Variable Breakdown 55
Using Variables at the Keyboard 56
The LET Statement 56
Implied LET 56
Array Variables 57
Defining the size of an array 58
Implicit Definition 58
Array Elements 58
Array Identifier 58
Declaring and Dimensioning Variables 59
Subscripts 59
The OPTION BASE Statement 60
The DIM Statement 60
The INTEGER Statement 61
The SHORT Statement 62
The REAL Statement 62
The COM Statement 63
Redimensioning an Array 64
The REDIM Statement 64
More Ways to Assign Values to Variables 65
The READ and DATA Statements 66
The MAT READ Statement 66
DATA Pointer 67
The RESTORE Statement 68
The INPUT Statement 69
The MAT INPUT Statement 71
Storage of Variables 73

Chapter 6: Array Operations

Assigning a Constant Value 76
1. MAT...CON 76
2. MAT...ZER 76
3. MAT-Initialize 77
The MAT - Copy Statement 78
Mathematical Operations 79
Scalar Operations 79
Arithmetic Operations 80
Functions 81
Matrices and Vectors 82
MAT...IDN 82
Matrix Multiplication 83
MAT...INV 85

MAT...TRN 87
MAT...CSUM 87
MAT...RSUM 88
Array Functions 89
SUM Function 89
ROW Function 89
COL Function 89
DOT Function 90
DET Function 90

Chapter 7: String Operations

Overview 94
Dimensioning a String 94
Explicit Dimensioning 94
Implicit Dimensioning 95
String Arrays 95
String Expressions 96
Substrings 96
String Concatenation (&) 97
Assigning a Value to a String 98
The LINPUT Statement 98
The EDIT Statement 99
String Variable Modification 100
No Substring Specifers 101
One Substring Specifier 101
Two Substring Specifiers 102
The Null String 103
String Functions 103
The LEN Function 103
The POS Function 104
The VAL Function 105
The VAL$ Function 105
The CHR$ Function 106
The NUM Function 107
The UPC$ Function 107
The LWC$ Function 108
The RPT$ Function 108
The REV$ Function 109
The TRIM$ Function 109
Relational Operations 110
Variable Diagram 111
Memory Usage 111

Chapter 8: Branching and Subroutines

Unconditional Branching 114
The GOTO Statement 114
The ON...GOTO Statement 114
Summary 115
The IF...THEN Statement 115
The FOR and NEXT Statements 117
Nesting 120
FOR-NEXT Loop Considerations 121

Subroutines 122
The GOSUB Statement 122
The ON...GOSUB Statement 123
Summary 124
The DEF FN Statement 125
Summary 126

Chapter 9: Subprograms
Why Use Subprograms? 128
Types of Subprograms 128
Terms 129
Parameters 129
Formal Parameters 129
Pass Parameters 130
Passing the Parameters 130
Summary 133
Multiple-Line Function Subprograms (DEF FN) 134
Subroutine Subprograms (SUB and CALL) 136
Subprogram Considerations 138
What Happens 138
Using the COM Statement 138
Variable Allocation Statements 140
Local Variables 140
Speed Considerations 140
Files 141
Editing Subprograms 142

Chapter 10: Output
The BEEP Statement 144
The DISP Statement 144
Printed Output 146
The PRINTER IS Statement 146
The PRINT Statement 147
Output Functions 149
The TAB Function 149
The SPA Function 150
The LIN Function 151
The PAGE Function 152
The MAT PRINT Statement 153
The PRINT USING and IMAGE Statements 155
Format String 155
Reusing the Format String 155
Delimiters 156
Blank Spaces 156
String Specification 156
Numeric Specification 157
Digit Symbols 157
Radix Symbols 159
Sign Symbols 160
Digit Separator Symbols 161
Exponent Symbol 162
Floating Symbols 162
Replication 163

Compacted Specifier 164
Carriage Control 164
Field Overflow 165
Summary 166
Considerations 166
Advanced Printing Techniques 166
Overlapped Processing 166
The OVERLAP Statement 167
The SERIAL Statement 167
Accessing Color on the CRT 167
Color Using CONTROL 167
Color Using CHR$ 167
Color Using the Escape Code Sequence 168

Chapter 11: Mass Storage Operations
Terms 170
The MASS STORAGE IS Statement 172
Structure 173
Files 173
Records 173
EOF's and EOR's 174
Physical Records 174
End-of-File and End-of-Record Marks 174
The Directory 175
Tape Cartridge Directory 175
The INITIALIZE Statement 176
The CAT Statement 177
The CAT TO Statement 179
Storing and Retrieving Programs 181
The SAVE Statement 182
The GET Statement 182
The LINK Statement 184
The RE-SAVE Statement 184
The STORE Statement 185
The LOAD Statement 186
The RE-STORE Statement 186
Storing and Retrieving Data 187
Considerations 187
The CREATE Statement 187
The ASSIGN Statement 188
Serial File Access 189
The PRINT# Statement - Serial 189
The READ# Statement - Serial 191
Random File Access 193
The PRINT# Statement - Random 193
The READ# Statement - Random 194
Repositioning the Pointer 194
The MAT PRINT# and MAT READ# Statements 194
Random vs. Serial Method 195
Closing a File – ASSIGN 196
Other Data File Operations 196
TYP Function 196
The ON END# Statement 197

The OFF END# Statement 198
EOR Errors 198
Data Storage 198
The BUFFER Statement 199
The CHECK READ Statement 200
The CHECK READ OFF Statement 201
The PROTECT Statement 201
The PURGE Statement 202
The COPY Statement 202
The RENAME Statement 203
STORE KEY and LOAD KEY 203
STORE BIN and LOAD BIN 204
STORE ALL and LOAD ALL 204
The Tape Cartridge 205
Recording on the Tape 205
Write Protection 205
Inserting and Removing the Tape Cartridge 206
General Tape Cartridge Information 206
The REWIND Statement 207
Mass Storage Errors 207
Optimizing Tape Use 207

Chapter 12: Editing and Debugging

Debugging a Program 210
The TRACE Statement 210
The TRACE WAIT Statemen 211
The TRACE PAUSE Statement 211
The TRACE VARIABLES Statement 212
The TRACE ALL VARIABLES Statement 212
The TRACE ALL Statement 213
The NORMAL Statement 213
Error Testing and Recovery 213
The ON ERROR Statement 213
Error Functions 214
The OFF ERROR Statement 215

Chapter 13: Special Function Keys

Pre-defined definitions 218
Special Features 218
Typing Aids 220
The EDIT KEY Command 220
The SCRATCH KEY Command 225
The LIST KEY Command 225

Chapter 14: Program Interrupts

Introduction 227
Priority 227
Changing the System Priority 228
Scope of Interrupt Statements 228
How Interrupts Interact 228
ON KEY, ON KBD, and ON INT 228
ON ERROR and ON END 229

Errors 229
When are Interrupts Active? 230
The DISABLE Statement 230
The ENABLE Statement 230
The ON KBD Statement 230
Priority 231
ALL 231
ON KBD Buffer 231
Considerations 231
KBD$ Function 232
The OFF KBD Statement 234
The ON KEY# Statement 234
Priority 234
Considerations 237
The OFF KEY Statement 237
Summary 238
Softkeys 238

Appendix A: Advanced Printing Techniques

Introduction 239
CRT Memory 239
CRT Special Features 239
Using Control Codes 241
CRT vs. Printer 242
Considerations 242
Disabling Control Codes 243
CRT Selective Addressing 243
Introduction 243
The Cursor 244
Adressing Schemes 244
Setting the Cursor Position 245
Absolute Addressing 245
Relative Addressing 246
Combining Absolute and Relative Addressing 246
Moving the Cursor 247
Using Tabs 247
Clearing, Inserting and Deleting Lines 258
Inserting and Deleting Characters 248
Rolling the Display 248
Selective Scrolling (Memory Lock) 249
The Internal Printer 250
Structure 250
Rows Per Line 250
Margins 251
Setting Tabs 251
New Characters 252
String Replacement 255
150% Size Characters 256
Underlining 256
Plotting Mode 257
Summary of Escape Codes 258
Examples 258

Appendix B: Programming Exercises 263

Reference Tables

Glossary. 1
ASCII Character Codes 10
Roman Extension Character Codes 11
Metric Conversion Table 12
Reset Conditions 13
Memory 14
System 45 Compatibility 19
Graphics Firmware Differences 24

Error Messages

Subject Index

Chapter 1

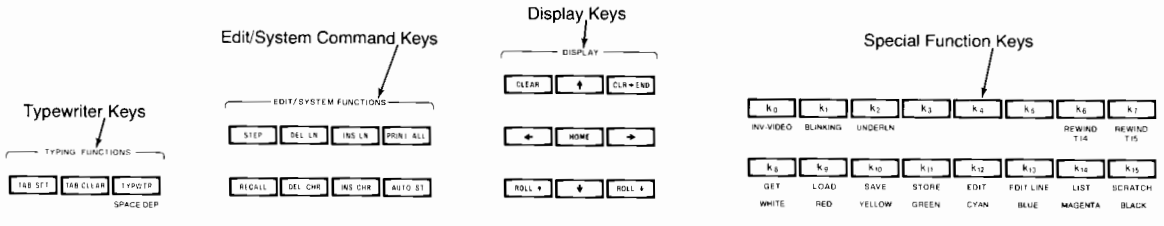
General Information

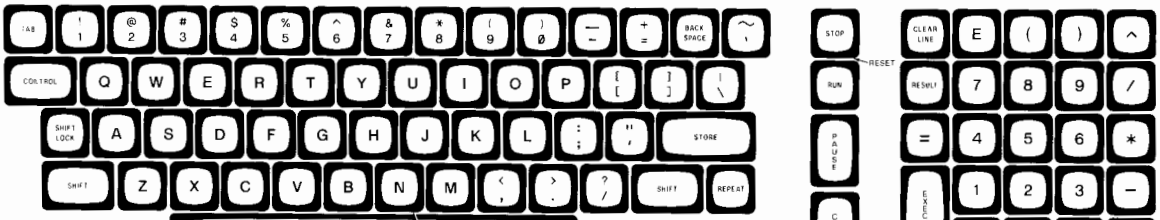
Your 9845 Desktop Computer is a high speed, versatile computing tool. You can use it to perform calculations and to enter and run programs written in BASIC (Beginner's All-purpose Symbolic Instruction Code). The 9845 is designed for both the programmer and the system operator since it can be used for interactive writing and debugging of programs and for entering data into a running program.

If you have just received your computer, please refer to the Installation, Operation and Test Manual for information about initial set-up and operation procedures. Otherwise, you can begin using the computer by setting the power switch on the right-hand side of the machine to the "1" position and waiting for 9845 READY FOR USE to be displayed.

The Keyboard

The keyboard is the primary means for entering programs and data into the computer. This section provides information about the functions of many of the keys. For more information about a specific key, refer to the index.





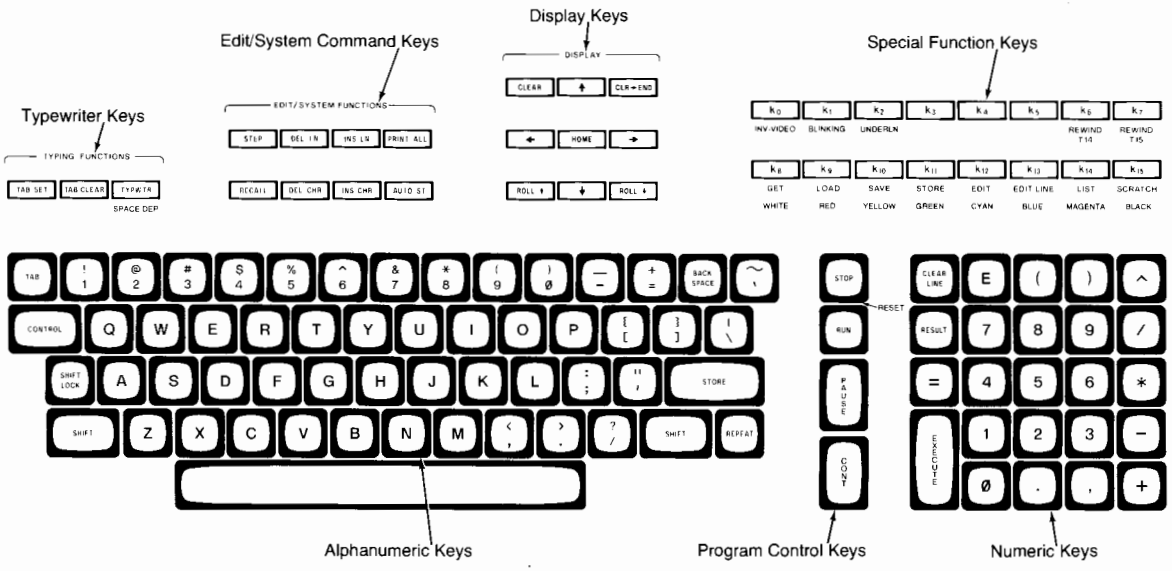
Chapter 1
General Information

Your 9845 Desktop Computer is a high speed, versatile computing tool. You can use it to perform calculations and to enter and run programs written in BASIC (Beginner's All-purpose Symbolic Instruction Code). The 9845 is designed for both the programmer and the system operator since it can be used for interactive writing and debugging of programs and for entering data into a running program.

If you have just received your computer, please refer to the Installation, Operation and Test Manual for information about initial set-up and operation procedures. Otherwise, you can begin using the computer by setting the power switch on the right-hand side of the machine to the "1" position and waiting for 9845 READY FOR USE to be displayed.

The Keyboard

The keyboard is the primary means for entering programs and data into the computer. This section provides information about the functions of many of the keys. For more information about a specific key, refer to the index.



Alphanumeric and Numeric Keys

Use these keys to enter letters, numbers and other characters. The alphanumeric keys work like those on a typewriter except that, normally, pressing a key gives you an uppercase letter, while holding down SHIFT and pressing a key gives you a lowercase letter. To reverse this and make the keys work like a typewriter, press the TYPWTR key. The E key in either the numeric or alphanumeric section can be used to enter an E in a number. The E indicates that an exponent follows, when expressing a number in scientific notation.

General Purpose Keys

Key	Function
EXECUTE	Perform the operation (numeric computation, command or statement without a line number) that has been entered.
RECALL	Return any keyboard entry that was followed by STORE, CONTINUE, or EXECUTE to the keyboard entry area. Entries are stored into a 1296-byte (character) buffer on a last-in, first-out basis. Press RECALL to recall a previous keyboard entry. To move the other direction through the recall buffer (recalling more-recent entries) press RECALL while holding down SHIFT. When the recall buffer is full, each new entry causes one or more of the oldest entries, depending on size, to be lost.
RESULT	By pressing RESULT, the answer of the most-recently executed keyboard calculation can be used in another calculation. You can also type in RES to use this **result function** .
PRT ALL	When this key is latched, keyboard operations and system messages are logged. There is more information later in this chapter about logging keyboard entries.
AUTOST	If this key is latched when the computer is switched on, the computer attempts the following operation on the right-hand tape drive: LOAD "AUTOST",1. This allows the computer to automatically load and run a program.
CONTROL	This key is used with several other keys, such as TYPWTR, to provide special functions. These functions are discussed in this manual when appropriate.

Typing, Line Editing and CRT Display Keys

Key	Function
REPEAT	Pressing REPEAT simultaneously with another key causes that key to be rapidly repeated.
TAB SET	Sets a tab at the current position of the cursor, like on a typewriter.
TAB CLR	Clears a tab at the position of the cursor. All tabs are cleared at power-on, reset and SCRATCH A.
TAB	Moves the cursor to the next tab setting without changing any characters it moves across. If no characters have been keyed in, intervening character positions are filled with spaces (blanks). If there are no further tab settings and TAB is pressed, the cursor moves to the last (160th) character position in the keyboard entry area; a beep occurs if the cursor was to the left of position 148 prior to the tab. Tabs can be useful for inserting comments at the end of program lines. You can easily line up all of your comments by setting a tab somewhere between columns 40 and 60.
TYPWTR	Sets the keyboard to **typewriter mode** to make the keyboard identical to a typewriter in that uppercase letters are obtained when the SHIFT key is pressed and lower case is obtained without shifting. When this key is pressed, TYPWTR is displayed on the right-hand side of the system comments line. To exit typewriter mode, press TYPWTR again. Typewriter mode is programmable.

CLEAR LINE	Clears the keyboard entry area and the system comments line of everything except any indicators for typewriter or space dependent mode and the run light.
CLEAR	Clears the entire CRT of everything except any mode indicators, the run light and any INPUT, LINPUT, or EDIT prompt.
→	Moves the cursor one character position to the right. If the cursor is one position to the right of the last character in the line, pressing this key one more time moves it to the first position in the line. Pressing down firmly causes rapid repetition of cursor movement.
←	Moves the cursor one character position to the left. If the cursor is at the beginning of the line, pressing this key one more time moves the cursor to the character position after the last character in the line. Pressing down firmly causes rapid repetition of cursor movement.
BACKSPACE	Moves the cursor one position to the left, circling around to the end of the line when the cursor is at the beginning of the line.
HOME	Moves the cursor to the **home position** which is the first position in the keyboard entry area.
SHIFT/HOME	Moves the cursor to the character position immediately following the last character in the line.
INS CHR	Lets you insert characters to the left of the cursor and causes the insert cursor (an inverse video block) to appear over the character at the position of the cursor. The rest of the line moves to the right. The insert mode is exited by pressing INS CHR again, moving the cursor, or by pressing STORE, CONTINUE, or EXECUTE.
DEL CHR	Deletes the character at the position of the cursor. The cursor remains in the same position and the rest of the line moves one position to the left as each character is deleted.
CLR→END	Clears the keyboard entry area from the position of the cursor to the end. It also clears the system comments line of everything except any indicators for typewriter or space dependent mode and the run light.
↑	Moves ("scrolls") the lines in the printout area up one line. If any lines are below the displayed lines, pressing this key brings one line up into the bottom line of the printout area. Pressing down firmly causes rapid repetition.
↓	Pressing this key causes one line, if any, above the top line in the printout area to move into the top line; the lines all scroll down. Pressing down firmly causes rapid repetition.
ROLL ↑	Performs the same operation as up arrow except that the next 10 lines are scrolled.
ROLL ↓	Performs the same operation as down arrow except that the next 10 lines are scrolled.

Special Function Keys

The Special Function Keys (SFK's), marked k0 through k15, provide a variety of functions such as entering frequently used statements and variable names with one keystroke. Many of them have pre-defined definitions when the computer is turned on which are printed below the keys. These keys are covered in Chapter 13.

Resetting the Computer

If the computer becomes inoperative due to a system or I/O malfunction, you may need to reset it to return it to a ready state. This is done by holding down CONTROL, then pressing STOP. This is the reset operation. (Notice that RESET is indicated on the front of the STOP key.)

Reset immediately aborts all machine activity. It is a hardware-oriented operation and returns the computer as well as all peripherals and HP-IB interfaces to a ready state. If a program is running, any pending or executing I/O operation is terminated and data may be lost. You may also lose data if a mass storage device is being accessed.

NOTE

There is a slight possibility that the reset operation will cause the entire memory to be scratched as if you had executed SCRATCH A. Use reset only if nothing else, such as pressing STOP, brings the machine to a ready state.

Refer to the Reset Table in the Reference Tables section for a list of conditions affected by reset.

Logging Keyboard Operations

Use **print all mode**, to obtain a printed log of all operations that are executed from the keyboard, including computations, displayed results, trace messages and error messages. This provides a useful audit trail of previous operations for later reference – to duplicate a procedure for example. It is also useful for displays that are longer that 80 characters, like executing five computations separated by commas. Otherwise, only the last line is displayed.

Print all mode is set by pressing and latching the PRT ALL key, which is with the EDIT/SYSTEM FUNCTION keys. Print all mode is turned off by unlatching PRT ALL.

The PRINT ALL IS Statement

The log of keyboard operations is printed on the **print all printer**. This printer is the CRT when the computer is turned on and after SCRATCH A is executed. You can change it by executing the PRINT ALL IS statement, which can be done either in a program or from the keyboard.

```
PRINT ALL IS select code [ , HP-IB device address]
```

The select code is a number used to access an internal or external device. 16 is the select code of the CRT; 0 is the select code of the internal printer. The select code of an external device is set on the interface card that connects the device to the computer. The HP-IB device address is set on a device that is connected to the computer with an HP-IB interface card.

Here are some examples of the PRINT ALL IS statement –

```
PRINT ALL IS 16     ! CRT is print all printer
PRINT ALL IS 0      ! Internal printer is print all printer
PRINT ALL IS 6      ! Printer at select code 6
PRINT ALL IS 7,2    ! HP-IB printer
```


Using the Keyboard While a Program is Running

Live keyboard mode allows computations and most statements and commands to be executed from the keyboard while a program is running. You can even change a line of the running program by typing the new line and pressing STORE. You can check the value of a variable by typing its name and pressing EXECUTE. (However, if execution is currently in a subprogram, you may get an unexpected result if the variable isn't defined in that subprogram.)

To see how live keyboard mode works, key in the following program lines, press STORE after each one, then press RUN.

```
10   DISP "PROGRAM RUNNING"
20   GOTO 10
30   DISP "PROGRAM DONE"
40   END
```

While the program is running, you can use the numeric keys for computations, like balancing your checkbook. Now change the program while it is running by typing and storing this line –

```
20   GOTO 30
```

The SUSPEND INTERACTIVE Statement

Live keyboard can be disabled by executing the SUSPEND INTERACTIVE statement –

```
SUSPEND INTERACTIVE
```

While a program is running, any attempt to execute a keyboard operation or alter the program by storing a line or executing a program control command such as CONT causes a PROGRAM EXECUTING or SYSTEM BUSY message to appear. When live keyboard is disabled, [STORE] and [EXEC] are disabled as well.

Execute SUSPEND INTERACTIVE and change line 20 of the previous program back to `20 GOTO 10`. Now run the program and try to add 2+2 or store a program line.

The RESUME INTERACTIVE Statement

Live keyboard can be re-enabled by executing the RESUME INTERACTIVE statement –

```
RESUME INTERACTIVE
```

1

To do this while a program is running, you should press [PAUSE] first, then press [CONT] after you re-enable live keyboard.

Error Messages and Warnings

When an error occurs, the machine beeps and displays an error number or a warning message. The error number references a description that helps you pinpoint the cause of the error. For example, typing `5/0` [EXEC] causes `ERROR 31` to be displayed. `ERROR 31` indicates division by zero, an operation the computer can't perform. A warning message can also appear which describes the error. Executing `3*(5/7` causes `IMPROPER EXPRESSION` to be displayed with the expression in the keyboard entry area with the cursor indicating where the parenthesis should be.

If an error occurs within a running program, the machine halts and the line number in which the error occurred is displayed. For example, when

```
10 X=TAN(3*PI/2)
20 END
```

is executed, `ERROR 24 IN LINE 10` occurs, indicating TAN(n*π/2) where n is odd.

A complete list of the error numbers and their meanings is given in the Reference Tables, in the Quick Reference supplied with the computer, and also on the pull-out cards under the CRT.

9845A vs. 9845B/C

The HP 9845A/B/C are known as the System 45. Information concerning the differences between these mainframes is found in the Reference Tables.

Chapter 2
Programming Tutorial

The 9845 is designed to help you solve problems. You can solve simple problems by performing calculations from the keyboard. You can use one or more software programs provided by HP. Or you can write your own programs.

Programs can involve computations, output of results, user interaction, decision making and other tasks. They can be as simple or as complex as you need them to be. You tell the computer how to solve your problem; the computer performs the complex calculations, makes the output look good and reduces the time needed to solve the problem.

Programming a computer is a straightforward task with three basic steps:

1. Define the problem.
2. Plan the solution.
3. Translate the plan into a program.

This chapter is designed to familiarize you with these steps. It also provides an overview of many simple programming statements. If you already understand programming concepts, you can find information about the 9845's mainframe statements in the remaining chapters of this manual. If you need more information about programming, there are many good text books that teach programming and BASIC (Beginner's All-purpose Symbolic Instruction Code, the language that you use to program your 9845). Additionally, Hewlett-Packard offers a course in BASIC on desktop computers.

Before you read this chapter, you should be familiar with how to operate the 9845. If you are not, read Chapter 1 of the Installation, Operation and Test Manual and Chapter 1 of this manual.

Where Do You Go From Here?

This chapter is limited to the rudiments of programming. Once you feel comfortable with the ideas and statements, you can explore the rest of the chapters in this manual.

Problem Solving: Defining the Problem

The first step in using your computer to solve your problem is to **define the problem thoroughly**. Otherwise, you may not know if you have reached the solution. Here are some questions to ask yourself:

- What **exactly** do I want to achieve?
- What output do I want? some tables? a series of conclusions? some data computed, printed and stored away for later use?
- What data is necessary to produce the output? Is the data fixed or might it vary each time you run the program?
- What computations are needed?
- When the program makes a decision, what are all the alternatives?
- How should the program identify and handle potential errors?

Another factor to remember in problem definition is the fact that the computer is a machine. A computer does only what it is told, so you must tell it **everything** that you want it to do.

Example

Here is a simple problem that we will solve: convert a Fahrenheit temperature to a Celsius temperature and print both temperatures.

This is the same example used in Chapter 1 of the Installation, Operation and Test Manual. In this chapter, subsequent examples expand on this first one.

Problem Solving: Developing a Solution

Once your problem is defined, the next step in problem solving is to develop a solution to serve as a guide for writing your program. One way to develop a solution is to construct a **program outline**. Another method is to develop a **flowchart**, which is useful for picturing the flow of a complex program. As you program, you will find the method that works best for you.

Program Outlines

A program outline is written in plain English and is similar to a topical outline for writing. Once you have a precise problem definition, you begin by dividing your task into smaller tasks. Then divide these tasks into even smaller tasks. Continue this breakdown until each task is as simple as possible; these simplest tasks can even be close to BASIC statements. It may seem unnecessary to write all these simple steps, but simple details are essential to make the computer do what you want it to do. Additionally, defining the simplest tasks may point out a higher-level step that should be modified.

Program Outline Example

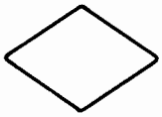
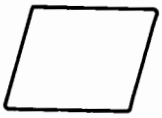
Our sample problem has a simple outline:

1. Generate a Fahrenheit temperature.
2. Calculate the Celsius temperature.
3. Print the values.
   a. Print text identifying the values.
   b. Print the values.

Note that steps 3a and 3b are the two steps needed to accomplish step 3; they aren't done in addition to step 3.

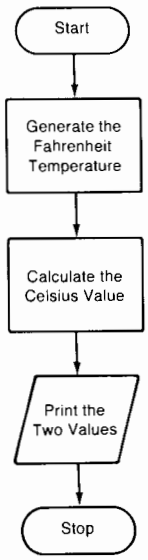
Flowcharts

A flowchart is a graphic representation of the steps for the program to take. It consists of various symbols with instructions in them and serves as a map to the solution. However, a flowchart normally doesn't include details such as output or variable names. The instructions are not program statements; they are general descriptions of the steps. Here are the basic flowchart symbols.

Symbol	Meaning
	Terminator: signifies the beginning or end of the program.
	Action Symbol: signifies processing, such as an arithmetic operation, that isn't represented by any other flowchart symbol.
	**Decision:** signifies alternate branches to two or more points in the flowchart based on the result of a decision.
	**Input/Output:** signifies any operation to an input or output device such as a printer.
	Flowline: shows the direction of the flow between symbols in the flowchart. If the arrowhead is omitted, flow is assumed to be from left to right and top to bottom.
	Connector: signifies an exit to or entry from some other part of the flowchart. It substitutes for a flowline when the direction of flow is broken.

Flowchart Example

Here is the flowchart for our sample problem.



Problem Solving: Writing the Program

Once you have planned a solution for your problem, your last task is to translate this plan into the computer's language. Each step in the plan becomes one or more **statements**. You put one statement on each line with a **line number** at the beginning of the line and press the STORE key to place the line in memory. When all the lines in the program are stored, press the RUN key to run the program. The computer then executes each statement in the program sequentially, unless the program specifies a change in program flow.

Programming statements fall into several general categories. The following list covers some simple BASIC statements. The 9845 uses an expanded version of this list in the mainframe and option ROMs.

For **assigning values to variables**, use LET, INPUT, LINPUT, READ and DATA. The LET statement is valuable; you can use it to perform **calculations** and use **functions** like SIN or TAN.

For **outputting results**, use DISP, PRINT, PRINT USING and IMAGE. Use PRINTER IS to specify where you want results printed. STANDARD, FIXED and FLOAT help you format numbers.

For **decision-making and controlling the line-to-line flow of the program**, use IF...THEN, FOR/NEXT, GOTO and GOSUB.

Of course, there are many other statements and built-in functions. They are covered in the remaining chapters of this manual.

Example

Our sample program is simple, requiring only five lines.

```
10    LET F_temp=155              ! Assign a value to the variable F_temp
20    C_temp=(F_temp-32)*(5/9) ! Calculate and assign a value to C_temp
30    PRINTER IS 16               ! Specify the CRT for output
40    PRINT F_temp;"degrees Fahrenheit = ";C_temp;"degrees Celsius."
50    END                                 ! Indicate the end of the program

155 degrees Fahrenheit=68.3333333334 degrees Celsius.
```

Here are some things to note:

1. Each statement has a line number. Type and execute EDIT to enable you to enter a program with line numbers automatically provided and incremented by 10.
2. Lines 10 and 20 are both LET statements (LET need not be included). LET assigns values to **variables**. A variable is a location in memory that you access with a unique name, such as F_temp. Rather than using simple names like X, Y and Z, you can use up to 15 characters to create meaningful names. Chapter 5 provides more information about variables.
3. Formulas and calculations must be put into linear form for the computer to solve. For example,

$$\frac{\frac{7}{6+5} - \frac{4x9}{11}}{81.35 + 72.66}$$

   would be entered as: (7/(6+5)−(4*9)/11)/(81.35+72.66). Parentheses and mathematical priority help determine the form; they are covered in Chapter 4.
4. The PRINTER IS 16 statement specifies that the data in the PRINT statement be printed on the CRT. PRINTER IS 0 specifies the internal printer. A PRINTER IS statement is not needed before every PRINT statement; include one only when you need to set or change the printer.
5. An END statement should be included to indicate the end of the program.
6. You can put a **comment** on the line by putting an exclamation mark (!) after the statement. Anything after that is considered a comment and is ignored when when the program is running. You can also put just the exclamation mark and comment after the line number. Comments enable you to put documentation with the program. Documentation is essential if others use your program or if you set your program aside for a time, then go back to it.

More Examples

Example Two

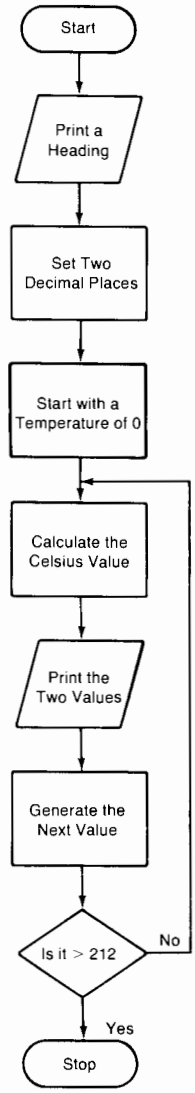
Let's expand the first problem to print a table of Fahrenheit to Celsius conversions from 0 ° to 212 ° Fahrenheit on the internal printer. Additionally, modify the output so that there are only two decimal places. (You find you don't need the accuracy of 10 decimal places.) The program outline and flowchart are:

1. Generate the Fahrenheit temperature, starting with 0.
2. Calculate the Celsius value.
3. Print the two values.
   a. Print a heading for the table.
   b. Set two-decimal-place accuracy.
   c. Print the two values.
4. Generate the next value.
5. If there is another conversion to make, go back to step 2.

As this outline is developed, we find one lower-level task that makes us modify a higher level. Steps 3a and 3b (print a heading and set the accuracy) will be repeated for every value. Since we only want one heading on the table and need set the accuracy only once, we modify the plan to be:

1. Generate the Fahrenheit temperature, starting with 0.
2. Prepare the table.
   a. Print the heading.
   b. Set two-decimal-place accuracy.
3. Calculate the Celsius value.
4. Print the two values.
5. Generate the next value.
6. If there is another conversion to make, go back to step 2.

The flowchart is shown on the right-hand side of the page.



The program now looks like:

```
10    First_temp=0
20    Last_temp=212
30    PRINTER IS 0
40    PRINT "Fahrenheit";TAB(20);"Celsius"! Print the table heading
50    FIXED 2                             ! Set two decimal places
60    FOR Temp=First_temp TO Last_temp    ! FOR and NEXT form a LOOP
70       C_temp=(Temp-32)*(5/9)
80       PRINT TAB(4);Temp;TAB(20);C_temp ! TAB makes the right spacing
90    NEXT Temp                           ! NEXT increments Temp by 1
100   END
```

Here is part of the output.

```
Fahrenheit          Celsius
    0.00            -17.78
    1.00            -17.22
    2.00            -16.67
    3.00            -16.11
    4.00            -15.56
    5.00            -15.00
    6.00            -14.44
    7.00            -13.89
    8.00            -13.33
    9.00            -12.78
    10.00           -12.22
    11.00           -11.67
    12.00           -11.11
    13.00           -10.56
    14.00           -10.00
    15.00           -9.44
    16.00           -8.89
    17.00           -8.33
    18.00           -7.78
    19.00           -7.22
    20.00           -6.67
    21.00           -6.11
    22.00           -5.56
    23.00           -5.00
    24.00           -4.44
    25              -3.89
```

Here are some things to note about this new program.

1. To enter this new program, execute EDIT, then edit the first program. Use the INS LN and DEL LN keys to insert and delete lines. Execute REN to renumber the lines by 10.
2. FOR and NEXT keep you from having to type the calculation 213 times. You specify initial and final values (First_temp and Last_temp) for the **loop counter**, which is Temp in this case. The computer automatically increments Temp and repeats the loop the proper number of times.
3. The TAB function used in lines 40 and 80 moves the printed output over so that the heading and the columns align nicely.

Example Three

This example has three things added to the problem definition:

- Omit the two decimal places on the Fahrenheit temperature; they aren't needed.
- Allow the program user to specify how many values are converted – every one, every other one, every third one, etc.
- Store all the Fahrenheit and Celsius values onto a tape cartridge for later use. Use an array to hold the values until all the conversions are done. (An array is a collection of variables with one name.)

The new program and part of the output are shown below.

```
10     DIM Conv_table(212,1:2)            ! Set up a 213 by 2 ARRAY
                                            0 to 212; 1 to 2
20     DIM Heading$[30]                   ! Set up a 30-character STRING
30     PRINTER IS 0
40     DEF FNTemp_conv(X)=(X-32)*(5/9)    ! Define a FUNCTION
50     Heading$="Fahrenheit          Celsius"
60     PRINT Heading$
70     INPUT "Type the value by which to count the table, press CONTINUE"
,Skip
80     FOR Temp=0 TO 212 STEP Skip        ! Increment Temp by Skip
90        C_temp=FNTemp_conv(Temp)        ! Use the defined function
100       Conv_table(Row,1)=Temp          ! Put values into the array
110       Conv_table(Row,2)=C_temp
120       STANDARD                        ! No decimal places
130       PRINT TAB(4);Temp;              ! Semicolon prevents line feed
140       FIXED 2                         ! Two decimal places
150       PRINT TAB(20);C_temp            ! Print the rest of the line
151       Row=Row+1
160    NEXT Temp                          ! NEXT increments Temp by Skip
180    ! ****************************************************************
190    ! The following lines create a file on a tape in the righthand
200    ! drive and store the array containing the conversion table into
         the file.
210    !
220    MASS STORAGE IS ":T15"             ! Put the file on a tape
230    CREATE "TFILE",15                  ! Create a file with 15 256-byte
                                            records
240    ASSIGN #1 TO "TFILE"               ! Open the file
250    PRINT #1;Conv_table(*)             ! Store the array in the file
260    ASSIGN #1 TO *                     ! Close the file
270    END
```

```
Fahrenheit          Celsius
    0               -17.78
    3               -16.11
    6               -14.44
    9               -12,78
   12               -11.11
   15               -9.44
   18               -7.78
   21               -6.11
   24               -4.44
```

Here are some things to note about this new program.

1. Lines 10 and 20 reserve space in memory for an array and a string variable. A string variable differs from a numeric variable in that its value consists of characters. You must dimension a string if you want to put more than 18 characters in it. Chapter 7 contains information about strings.
2. The array is a two-dimensional array with 213 rows and two columns, specified in the DIM statement by (212,1:2). 212 specifies the upper bound of the first dimension. The lower bound is zero unless OPTION BASE 1 is specified. 1:2 specifies both the lower and upper bounds of the second dimension.
3. Line 40 defines a **user-defined function** that is used like built-in system functions such as SIN. Line 90 shows the function being called. The value for Temp in line 90 is substituted for X in the formula in line 40. Built-in functions are covered in Chapter 4. User-defined functions are covered in Chapters 8 and 9.
4. The INPUT statement in line 70 allows the program user to specify the value for a variable – Skip in this case. The text specified in quotes is displayed to prompt the user for the value he should enter.
5. In line 80, Skip is used to specify a step or increment value for the FOR/NEXT loop to use instead of 1.
6. Lines 100 and 110 store the Fahrenheit and Celsius values into the array for storage later onto a tape.
7. The semicolon after Temp in line 130 suppresses the normal linefeed so that the Celsius value printed with two decimal places (line 150) is on the same line as the Fahrenheit value printed by line 130. Try removing the semicolon and running the program.
8. Lines 220 through 260 store the table onto a tape cartridge. MASS STORAGE IS specifies that subsequent mass storage operations be directed to the right-hand tape drive. It is wise to specify this in case previous programs or users have specified a different mass storage device. The CREATE statement sets up a file with 15 256-byte records. **Note:** If you run this program more than once, use a different tape cartridge, change the file name or delete line 230; the tape cannot have the same file name used more than once. ASSIGN# opens and lets you access a file; it can also close the file. PRINT# copies the values in the array into the file.

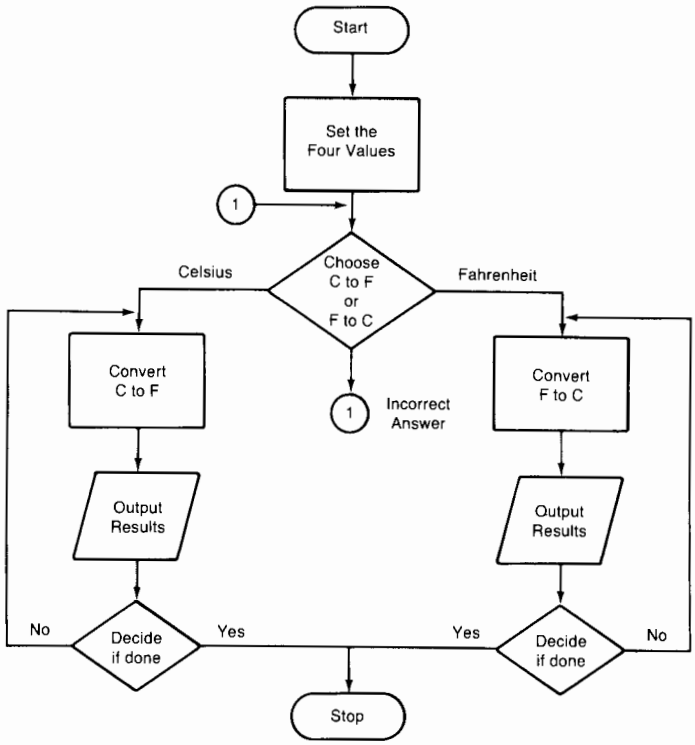
Example Four

This example has a slightly different purpose: Given four values, have the program user specify either a Fahrenheit-to-Celsius or a Celsius-to-Fahrenheit conversion, then perform the conversions and print the original and converted values.

The program outline is:

1. Specify the four values to be converted.
2. Ask the user which conversion he wants to perform.
3. Check for a proper response; if the response is improper, go to step 2.
4. Perform the desired conversion.
5. Print the original and converted values.
6. If there is another conversion, go to step 4.

The flowchart is:



The program and output are:

```
10     PRINTER IS 0
20     ! ***************************** Data for the conversion:
30     DATA 0,50,100,212                  ! Four numeric values
40     ! ***************************** Decide which conversion:
50     INPUT "Choose a conversion: 1 for C-to-F; 2 for F-to-C; press CONTINUE",N
60        IF N=1 THEN C_to_f          ! Specify a line label or line number
70        IF N=2 THEN F_to_c             ! GOTO is implied
80     BEEP                              ! Get the user's attention
90     PRINT "MUST ENTER 1 OR 2 !!! TRY AGAIN !!!"
100    GOTO 50                           ! Back to the INPUT statement
110    !
120 F_to_c:    ! ****************************** N=2; Fahrenheit to Celsius:
130    PRINT "Fahrenheit to Celsius:",LIN(1)     ! LIN prints a blank line
140    FOR Counter=1 TO 4
150       READ Temp              ! Read a value for Temp from DATA statement
160       New_temp=(Temp-32)*(5/9)
170       PRINT Temp;"degrees Fahrenheit = ";New_temp;"degrees Celsius"
180    NEXT Counter
190    STOP     ! **************** STOP stops the program in the middle
200    !
210 C_to_f:    ! ******************************** N=1; Celsius to Fahrenheit
220    PRINT "Celsius to Fahrenheit:",LIN(1)
230    FOR Counter=1 TO 4
240       READ Temp
250       New_temp=32+9/5*Temp
260       PRINT Temp;"degrees Celsius = ";New_temp;"degrees Fahrenheit"
270    NEXT Counter
280    END
```

```
Celsius to Fahrenheit
0 degrees Celsius = 32 degrees degrees Fahrenheit
50 degrees Celsius = 122 degrees Fahrenheit
100 degrees Celsius = 212 degrees Fahrenheit
212 degrees Celsius = 413.6 degrees Fahrenheit
```

Chapter 3
Programming

page 19 • **EDIT LINE** (lets the program be entered and edited)
page 21 • **DEL** (deletes selected program lines)
page 22 • **AUTO** (numbers lines automatically as they are entered and stored)
page 22 • **REN** (renumbers the program)
page 25 • **REM** (inserts non-executable remarks into the program)
page 26 • **LIST** (lists all or part of the program)
page 27 • **RUN** (starts execution of the program)
page 28 • **PAUSE** (suspends program execution)
page 28 • **CONT** (resumes program execution)
page 29 • **STOP** (stops the program – a logical end)
page 29 • **END** (stops the program – the physical end)
page 30 • **SCRATCH** (erases all or part of memory)
page 30 • **SECURE** (prevents selected program lines from being listed)
page 31 • **WAIT** (delays the program for a specified time)
page 31 • **TYPEWRITER ON/OFF** (sets/unsets typewriter mode)

Terms

- Program – a set of statements that lets your computer perform a task for you. Statements are preceded by a line number between 1 and 32 766 and can be up to 160 characters long.
- Edit line mode – used to enter and edit programs.
- Space dependent mode – makes program entering easier because variables and labels can be typed in all capital letters. Access with [CONTROL] - [TYPWTR].
- Comment delimiter – !; for inserting remarks at the end of a program line.
- Run light – ▒; displayed on the righthand end of the system comments line when a program or operation is executing.

Keys

[STORE] – enters a program line into memory.

[RUN] – runs the program.

[INS LN] [DEL LN] – insert and delete lines while in edit line mode.

[STEP] – runs the program one line at a time.

[PAUSE] – suspends program execution.

[CONT] – resumes execution where it was halted.

[STOP] – aborts the program and any I/O.

Syntax Conventions

The following conventions are used in the statement and command descriptions found in the 9845 manuals.

dot matrix – All items in dot matrix must appear exactly as shown.

[ ] – Items within solid square brackets are optional. Brackets in dot matrix are part of the statement.

... – Three dots indicate that the previous item can be repeated.

| – A vertical line between two parameters means "or"; only one of the two parameters can be included.

/ – A slash between two parameters means "and/or"; either or both of the parameters can be included.

Programming Terms

The following terms are used throughout the manual in the descriptions of the language.

Statement – A statement is an instruction to the computer that is assigned a unique line number, stored, and executed from a program. Most statements can also be executed from the keyboard without a line number. A statement is made up of one or more keywords and expressions.

Keyword – A keyword is a word that has a special meaning in BASIC, like PRINT, FOR and IF, and specifies an operation to be performed or the type of information in the statement. A **secondary keyword** is a keyword which isn't the first item in a statement such as THEN in an IF...THEN statement. Functions and logical operators are secondary keywords.

Command – A command is an instruction to the computer which is executed from the keyboard. Commands are executed immediately, do not have line numbers and can't be used in a program. They are used to manipulate programs and for utility purposes, such as listing key definitions.

Constant – A fixed numeric value within the range of the computer; for example, 29.5 or 2E12.

Character – A letter, number, symbol or ASCII control code; any arbitrary 8-bit byte defined by the CHR$ function.

Text – Any combination of characters, for example "ABC".

Name – A capital letter followed by 0 through 14 lowercase letters, digits or the underscore character. Names are used for variable names, labels, function names, and subprograms.

Line number – An integer from 1 through 32 766. Line numbers are arranged in ascending order, but you can type main program lines in any order because they are sorted as they are stored. In most cases, when a line number is specified, but is not in memory, the next highest line is accessed.

Label – A unique name that can be given to a program line. It follows the line number and is followed by a colon. In this example, Show_results is the label –

```
50 Show_results:   PRINT A,B,C
```

Line Identifier – A program line can be identified either by its line number or its label, if any. Line 50 above could be accessed with either –

```
100 GOTO 50
```

or

```
100 GOTO Show_results
```

Main Program – The central part of a program from which subprograms can be called is known as the main program. When you press (RUN), you access the main program. The main program can't be called by a subprogram.

Subprogram – A set of statements, separate from and after the main program, that performs a task under the control of the program segment that called it. Subprograms are covered in Chapter 9.

Program Segment – The main program and each subprogram are known as program segments.

Numeric Expression – A numeric expression is a logical combination of variables, constants, operators, functions, including user-defined functions, grouped within parentheses if needed. It has a single value.

Select Code – An expression (rounded to an integer) in the range zero through sixteen which specifies a setting on an interface card to an I/O device. The following select codes are reserved and can't be set on an interface –

- 0 Optional internal thermal printer and keyboard
- 13 Graphics option
- 14 Optional tape drive
- 15 Standard tape drive
- 16 CRT

HP-IB Device Address – An expression which specifies the HP-IB address that is set on a device. Its range is 0 through 30.

Program Fundamentals

A program is a set of instructions to the computer – an ordered set of statements. Each statement in a program **must** be preceded by a unique line number in the range 1 through 32 766.

Program lines can be up to 160 characters long including the line number and any label. After each line is typed in, you enter it into memory by pressing (STORE).

Pressing (STORE) also causes the line to be checked for syntax errors before it is stored. If there is a syntax error, the computer beeps and displays a message explaining the error. When the line is checked for syntax, parentheses may be added into expressions. This may cause the line to exceed 160 characters. It runs properly, but when listed, an asterisk appears after the line number and before the truncated line.

Normal program execution proceeds from the lowest-numbered line to the highest-numbered line. While a program is running (or any keyboard operation is being performed), the **run light** – ▒ – is displayed on the righthand end of the system comments line and an internal **program pointer** monitors which line is being executed.

Computer Museum

Entering Program Lines

There are three methods that can be used to enter line numbers and program lines. The first is to type the line number in manually before the statement. A second method is to use edit line mode to generate numbers as lines are stored. In the edit line mode, you can also insert, delete and change lines easily. The third way is to use the AUTO command to generate line numbers.

The EDIT LINE Command

The edit line mode lets you enter a new program or edit an existing one. It is entered by executing the EDIT LINE command –

EDIT[1] [LINE] [line identifier [, increment value]]

Examples

```
EDIT              ! Access lowest line in memory
EDIT 150          ! Access line 150
EDIT Routine,5    ! Access 'Routine'; increment
                    new lines by 5
```

When the EDIT LINE command is executed, the specified line, or the first line in memory if one is not specified, is displayed in line 12 of the CRT with the cursor after all the characters. Line 13 is also reserved for that program line. If you go into the edit line mode while a program is running, the program is paused. It is resumed automatically when you leave edit line mode. Here is a diagram of how the CRT looks in the edit line mode –

Lines 1-10	10 previous program lines (first 80 characters)
Line 11	blank
Lines 12 and 13	line being edited (keyboard entry area)
Line 14	System comments line
Line 15	blank
Lines 16-25	10 following program lines (first 80 characters)

1 There is a Special Function Key which is defined as "EDIT LINE". This key can be used to enter EDIT LINE.

The cursor can then be moved in the line and the line edited. When the line is the way you want it, press STORE. The next highest numbered line is then displayed in line 12.

To edit a different line, ↓, ↑, ROLL↓, and ROLL↑ can be used to move the program lines up or down. ↓ brings the next highest line into the editing line, while ↑ brings the previous line into the editing line. ROLL↓ and ROLL↑ cause the program to roll 10 lines in the specified direction.

Increment Value

After the end of the program is reached, the next line number is automatically generated. It is greater than the previous line number by the increment value or by 10 if the increment value isn't specified. The increment value must be a positive integer.

Automatic Indent

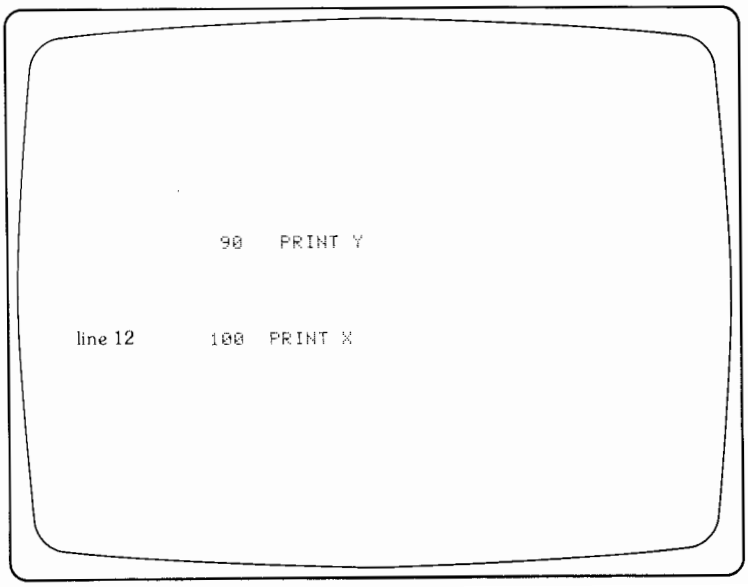
Using the edit line mode allows you to indent program lines automatically. This is possible when you are adding lines at the end of the program or inserting lines (discussed next). If you indent a line and store it, when the next line is generated, the cursor is indented as many spaces as it was in the previous line. The minimum automatic indent is six spaces from the left side.

Inserting Lines

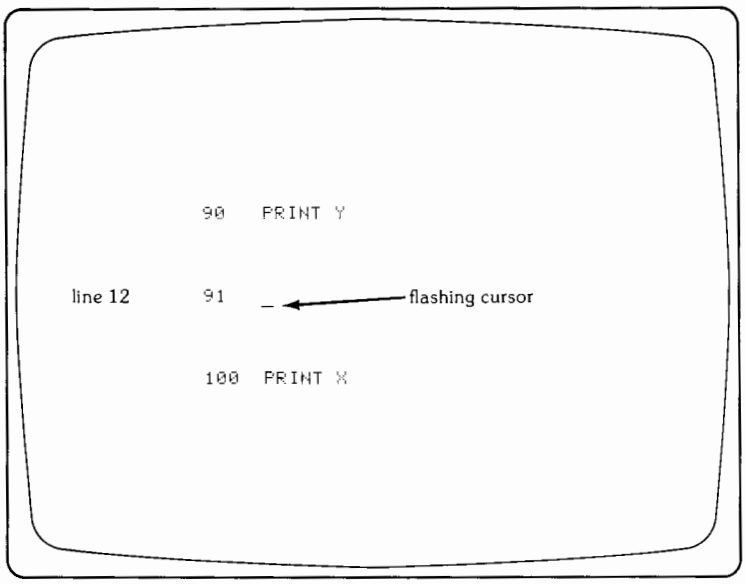
Lines can easily be inserted between existing program lines.

One way is to type in the line number and line, then press STORE. Another way is to use the insert line mode while you are in edit line mode. It is accessed by pressing INS LN.

Lines can then be inserted before the line which was in the keyboard entry area. A line number which is 1 greater than the previous line is generated and appears in line 12. For example, say the CRT looks like –



Pressing [INS LN] causes it to look like –



When line 91 is stored, line number 92 is generated. This continues until the insert line mode is exited by one of the following –

- Pressing [INS LN] again
- Pressing [DEL LN]
- Rolling the program with [↑], [↓]
- Changing the line number
- There is no more room between lines to insert another line. When this happens, the machine beeps and a warning appears –

OUT OF LINE NUMBERS

Deleting Lines

In the edit line mode, the line currently in line 12 is deleted from memory by pressing [DEL LN]. The next line is then displayed in line 12, and the rest of the lines scroll up.

The DEL Command

The DEL (delete) command is used to delete a line or section of a program when not in the edit line mode.

DEL first line identifier [, last line identifier]

If only one line identifier is specified, then only that line is deleted. Specifying two line identifiers causes that block of lines to be deleted. For example, to delete lines 40, and 100 through 150 from a program, execute –

```
DEL 40
```

and

```
DEL 100, 150
```

Exiting the Edit Line Mode

The edit line mode is exited by pressing [PAUSE]. [STEP] [CONT] [STOP] [RUN] [EXEC] or [CLEAR] can also be used.

The AUTO Command

The AUTO command allows lines to be numbered automatically as they are entered and stored. This saves you from having to type the line number each time you key in a statement.

AUTO [beginning line number [, increment value]]

If neither parameter is specified, executing AUTO causes line numbering to begin with 10 and to be incremented by 10 as lines are stored. If only the beginning line number is specified, the increment between line numbers is 10. Both the line number and the increment values must be positive integers. Automatic numbering halts when the keyboard entry area is cleared.

Examples

```
AUTO          ! Begin with 10, increment by 10
AUTO 100      ! Begin with 100, increment by 10
AUTO 100,5    ! Begin with 100, increment by 5
```

The REN Command

REN [beginning line number [, increment value]]

The renumber command causes the program in memory to be renumbered. This allows you to insert lines or to add more lines at the end. If no parameters are specified, the program is renumbered so that line numbering begins with 10 and is incremented by 10. If only the beginning line number is specified, the increment is 10.

Examples

```
REN          ! Begin with 10, increment by 10
REN 200      ! Begin with 200, increment by 10
REN 200,20   ! Begin with 200, increment by 20
```

When a program is renumbered, all line references (GOTO 50, for example) in the program are adjusted automatically to reflect the new line numbers, except for a reference to a non-existent line.

Spacing

In general, spacing between characters is arbitrary; the computer automatically sets proper spacing into each line as it is stored into memory. Only in text, REM statements, comments, and blanks after line numbers and labels does spacing remain exactly as input. These blanks allow lines to be indented.

Space Dependent Mode

The space dependent mode is useful for keying in a program that has long variable names. It causes spaces, or lack of them, between parts of a statement to become significant when entering program lines. In space dependent mode, variables, subprogram names and labels can be typed in all capital letters or in any combination of upper and lower case, as long as the first letter is upper case. Keywords **must** be separated from other parts of the statement by one or more blanks or by a delimiter like a comma or a #.

Space dependent mode is entered by holding down [CONTROL] then pressing [TYPWTR].[1] This causes the words SPACE DEPENDENT to appear on the right hand side of the system comments line.

1 Notice that SPACE DEP is indicated on the front of the TYPWTR key to show you that space dependent mode is accessed with [CONTROL] - [TYPWTR].

Example

Here is an example of how a program line may be typed in normal and space dependent modes –

Normal Mode –

```
10IFOutcome>PredictionTHENPRINTOutcome,Difference
```

Space Dependent Mode –

```
10 IF OUTCOME > PREDICTION THEN PRINT OUTCOME, DIFFERENCE
```


Both list identically –

```
10 IF Outcome>Prediction THEN PRINT Outcome,Difference
```

Here are some rules to follow when entering programs in space dependent mode –

- Any variable name that is the same as a secondary keyword **cannot** be entered in all capital letters.
- The label of a line that is the same as any keyword **cannot** be entered in all capital letters. However, when referenced, as in a GOTO statement, it can be entered in all capital letters, except after THEN.
- The first variable in an implied LET statement **cannot** be entered in all capital letters if it is the same as a keyword. This is also the case if the implied LET follows THEN.

Example

For example, in space dependent mode, trying to store –

```
10FORI=1TO10
```

gives an IMPROPER EXPRESSION message with the flashing cursor under the T. The computer is interpreting this as an assignment statement assigning the value 1 to the variable Fori.

When a program is listed after it was typed in space dependent mode, all names are converted to their normal format: capital letter followed by lower case.

To exit the space dependent mode, hold down [CONTROL] and then press [TYPWTR] again.

Space dependent and typewriter modes are mutually exclusive – if one is entered while the other is in effect, the new one cancels the old.

Remarks

Many times you may want to insert comments in order to make your program logic easier to follow. This can be done by using the REM (remark) statement or the comment delimiter !.

The REM Statement

REM [any combination of characters]

Remarks can be used to explain program lines or set off program segments.

Examples

```
10 Remark:    REM          A label can precede REM
20   REM  This section outputs all data
30   PRINT X
40   PRINT Y
50   REM
60   REM  You can say ANYTHING you want in a REM statement
```

Comment Delimiter

!, the comment delimiter, can be anywhere in a program line after the line number. If it comes immediately following the line number, it is just like a REM statement. **All** characters following a ! are considered part of a comment unless the ! is within quotes. The comment delimiter can also follow a command.

Using the comment delimiter, program lines and commands can contain comments.

Examples

```
10   !   This whole line is a comment   !!!
20   INPUT X,Y                 ! Request values
30   IF X>Y THEN 50            ! Branch if X>Y
40   STOP                      ! Execution stops
50   PRINT X,Y                 ! Output values
60   REM                       ! use TAB key to line up comments
70   END
```

The LIST Command

The LIST command is used to obtain a printed listing of the program or section of the program in memory. The listing is output on the CRT unless another device was specified as the standard printer with PRINTER IS. An asterisk between the line number and the line signifies that the line is longer than 160 characters and isn't listed completely.

LIST [beginning line identifier [, ending line identifier]]

If no parameters are specified, the entire program is listed. If one line identifier is specified, the program is listed from that line to the end. If two line identifiers are specified, that segment of the program, including beginning and ending lines, is listed.

Examples

```
LIST              ! Lists the entire program
LIST 50           ! Lists program beginning with line 50
LIST 200,250      ! Lists lines 200 through 250
```

Available Memory

When the listing is complete, the amount of unused memory available for use is displayed in the system comments line. So if you execute SCRATCH A then LIST, the number that is displayed is the total memory available for your use. This memory is expressed in bytes.

Alternate Printing Devices (LIST#)

The LIST command can be directed to a device other than the standard printer by specifying the select code of the alternate device.

LIST# select code [, HP-IB device address] [; beginning line identifier [, ending line identifier]]

Examples

```
LIST #16        ! List to CRT
LIST #0         ! List to internal printer
LIST #6         ! List to select code 6
LIST #6;50      ! List to select code 6--Line 50 on
LIST #6;15,55   ! List to select code 6--lines 15-55
LIST #7,2       ! List to HP-IB printer
```

The RUN Command and (RUN)

Program execution can be started at the lowest-numbered line by pressing (RUN).

It can also be started by executing the RUN command –

RUN [line identifier]

The line identifier must be in the main program and specifies that execution is to begin at that line; if no line is specified, execution begins with the first line in memory.

Examples

```
RUN            ! Begin at lowest-numbered line
RUN 150        ! Begin at line 150
RUN Routine    ! Begin at line labeled Routine
```

RUN causes a short pre-run initialization to occur which sets radian mode and the random number seed and also dimensions variables and initializes them to 0 or the null string. See the Reset Table in the Reference Tables for a complete list of items affected by RUN.

During the pre-run initialization, doubly defined labels and statements defined in ROMs which aren't present are detected and a warning message is given. However, functions defined in ROMs which aren't present are not detected.

After the pre-run phase, the program is executed.

The STEP Key

A program can also be run or continued by pressing (STEP).

When (STEP) is used, the program is executed one line at a time as (STEP) is pressed. The next line to be executed is displayed in the system comments line. When using (STEP) to run a program from the beginning, a pre-run initialization takes place the first time it is pressed. Pressing (STEP) a second time executes the first program line.

The PAUSE Statement and (PAUSE)

Execution can be suspended by pressing (PAUSE). The current line is completed and the program is halted at the next line to be executed; this line is displayed in the system comments line. Any current I/O operation is completed.

A pause can also be programmed using the PAUSE statement.

```
PAUSE
```

A useful application is to program a pause so that intermediate results can be checked and execution resumed. The PAUSE statement can't be executed from the keyboard. Press the PAUSE key instead.

The Continue Command and (CONT)

Program execution can be resumed where it was halted by pressing (CONT).

It can also be resumed by executing the CONT (continue) command –

```
CONT [line identifier]
```

The line identifier causes execution to resume at the specified line. If it is a line number that is not in memory, execution resumes with the next highest numbered line. CONT can also be used to start a program that was just run. No pre-run initialization takes place.

Execution of a paused program can also be restarted at the beginning with (RUN) or RUN.

Terminating Execution

All programs have a logical as well as a physical end. The logical end is that point where all statements have been executed the desired number of times and the program has completed the task for which it was designed. The physical end (highest-numbered line) of a program is the last (highest-numbered) line.

The STOP Key

Program execution can be halted before it is done by pressing (STOP).

When (STOP) is pressed, all I/O operations are aborted and data may be lost. The program pointer is reset to the first line of the main program. Don't use (STOP) to temporarily halt a program you'll want to resume. Use (PAUSE) instead.

Computer Museum

The STOP Statement

The STOP statement can be used to indicate the logical, rather than the physical end of a program –

```
STOP
```

Its purpose is to tell the computer to terminate execution of the program and reset the program pointer. It may appear at any point in the program. Some programs have several logical ends and so require several STOP statements. The STOP statement can't be executed from the keyboard. Press the STOP key instead.

The END Statement

The physical end (highest-numbered line) of a main program is indicated by the END statement –

```
END
```

END also terminates program execution and resets the program pointer to the lowest-numbered line. It is not mandatory to have an END statement as it is in other BASIC systems; however, it is good programming practice.

Reset

The reset operation (CONTROL-STOP) can also be used to stop a running program. All I/O operations are aborted and data may be lost.

It is also possible that the program and data can be destroyed just as if SCRATCH A had been executed. Therefore reset should not be used for stopping a program unless pressing (STOP) fails to halt the program.

The SCRATCH Command

The SCRATCH command is used to erase all or parts of memory; it can be used to erase programs, variables, keys, or the entire memory. [K15] is defined as a typing aid for SCRATCH at power on and SCRATCH A.

SCRATCH [P or V or C or KEY [key number] or [Kn] or A]

Command	Operation
SCRATCH	Erases program including DATA pointers.
SCRATCH A	Erases the entire memory. See the Reset table in Reference Tables.
SCRATCH C	Erases the values of all variables, including those in common.
SCRATCH P	Erases the program, variables, binary routines, DATA pointer and the files table.
SCRATCH KEY [key number]	Erases one or all SFK typing-aid definitions (but not control features).
SCRATCH V	Erases the values of all variables except those in common.
SCRATCH [Kn]	Erases the typing aid definition of the specified SFK.

3

The SECURE Statement

The SECURE statement is used to prevent selected program lines from being listed; instead, an asterisk appears after the line number. The secured lines execute normally, however.

SECURE [line identifier [, line identifier]]

If no line identifiers are specified, the entire program is secured. If one line identifier is specified, only that line is secured. Two line identifiers secure that block of lines, including the beginning and ending lines.

Examples

```
SECURE                  ! Secures all program lines
SECURE Formula          ! Secures the line labeled Formula
SECURE 100,150          ! Secures lines 100 through 150
```

There is no provision made for "unsecuring" a program, so be sure to specify the line identifiers accurately. However, a secured line can be deleted or replaced, and can be listed after that.

A program protected with SECURE can be reproduced onto a mass storage medium using STORE, but not using SAVE.

Miscellaneous Statements

The WAIT Statement

The WAIT statement is used to program a delay between the execution of two program statements –

```
WAIT number of milliseconds
```

The number of milliseconds is a numeric expression rounded to an integer in the range −32 768 through 32 767. A negative number defaults to a wait of zero. The delay specified by WAIT is only approximate.

The WAIT operation can only be interrupted by reset (CONTROL-STOP), which also stops the program.

The TYPEWRITER ON Statement

TYPWTR can be "pressed" from within a program to set the keyboard to typewriter mode, thus making input easier. This is done by executing the TYPEWRITER ON statement –

```
TYPEWRITER ON
```

When this statement is executed, the keyboard behaves just as if TYPWTR had been pressed.

The TYPEWRITER OFF Statement

Typewriter mode can be turned off from within a program by executing the TYPEWRITER OFF statement –

```
TYPEWRITER OFF
```

3

Chapter 4
Mathematics

Computer Museum

page 39 • **STANDARD** (sets standard format for output of numbers)
page 39 • **FIXED** (sets fixed-point format for output of numbers)
page 40 • **FLOAT** (sets floating point format (scientific notation) for output of numbers)
page 44 • **RANDOMIZE** (modifies the random number seed)
page 46 • **DEG** (sets degree mode for trigonometric operations)
page 46 • **RAD** (sets radian mode for trigonometric operations)
page 46 • **GRAD** (sets grad mode for trigonometric operations)
page 50 • **DEFAULT ON/OFF** (circumvents math errors with improper arguments by supplying default values. DEFAULT OFF cancels this process).

Functions

ABS (absolute value)
DROUND (digit round)
FRACT (fractional part)
INT (integer part)
MAX (maximum value)
MIN (minimum value)
PI (π)
PROUND (power-of-10 round)
RES (result of keyboard calculation)
RND (random number)
SGN (sign: +, −, 0)
SQR (square root)
EXP (Napierian e to a power)
LGT (common logarithm)
LOG (natural logarithm)
ACS (arccosine)
ASN (arcsine)
ATN (arctangent)
COS (cosine)
TAN (tangent)
SIN (sine)

Operators

+	=	AND	DIV (divide, return integer portion)
−	<	OR	MOD (modulo)
/	>	NOT	
*	< =	EXOR (exclusive or)	
∧	> =		
	< >		

Terms

- Range – the range of numbers that can be entered into your computer is $\pm 10^{-99}$ through $\pm 9.99999999999 \times 10^{99}$ and 0.
- Number format – the mode – STANDARD, FIXED or FLOAT – for output of numbers. STANDARD is set at power on and SCRATCH A. Internal form isn't affected.
- Angular units – the mode – DEG, RAD or GRAD – used for results and arguments of trigonometric functions. RAD is set at power on, SCRATCH A, RUN or reset.
- Hierarchy – determines the order in which multiple operations in an expression are performed.
- Parentheses – used to give a higher priority to lower-priority operations.

Operators

Arithmetic Operators

The arithmetic operations that can be performed on your computer are addition (+), subtraction (−), multiplication (*), division (/), exponentiation (^ or **; ** is listed ^), integer division (DIV), and modulo (MOD).

Relational Operators

Relational operators are used to determine the value relationship between two expressions. This can be especially useful for program branching if a specified condition is true.

Operator	Meaning
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
< > or #	Not equal to (either form is acceptable; it is listed < >)

The result of a relational operation is either a 1 (if the relation is true) or a 0 (if the relation is false).

Here are some examples of relational operations.

```
10    INPUT "ENTER THREE VALUES (A,B &C)",A,B,C
20    Logic=(A*B<C)
30    IF Logic THEN 60          ! Branch if A*B < C
40    PRINT "(A*B<C) is FALSE;  it's value is";Logic
50    GOTO 70
60    PRINT "(A*B<C) is TRUE;   it's value is";Logic
70    END
```

Entering the values 3, 4, 5 results in –

```
(A*B<C) is FALSE;  it's value is 0
```

The equals sign is also used in the assignment statement, as shown earlier in the chapter. In an assignment statement, the variable is to the left of the equals sign, the value is to the right. If the equals sign is used in such a way that it might be either an assignment or relational operation, the computer assumes that it is an assignment operation. For example, X=Y=Z assigns the value of Z to X and Y. X=(Y=Z) assigns the result of the operation Y=Z to X.

Logical Operators

The logical operators AND, OR, EXOR (exclusive or) and NOT are used for creating Boolean expressions. The expressions used with logical operators can be either relational or non-relational. If an expression is relational (like A<B), its true or false designation is determined by the particular relational value. If an expression is non-relational (like A), it is true if its arithmetic value is any value other than 0, and false if its arithmetic value equals 0.

The result of a logical operation is either 0 (false) or 1 (true). Logical operators are especially useful in determining whether or not certain sets of conditions are true.

Programming Hint

If you want to use a variable called T as the first operand in an AND or OR operation, put parentheses around the T when entering the line. This is to avoid confusion with the keywords TO and TAN. The computer takes out the parentheses after the line is stored.

AND Operator

numeric expression AND numeric expression

AND compares two expressions. If both expressions are true, the result is true (1). If one or both of the expressions is false, the result is false (0).

OR Operator

numeric expression OR numeric expression

OR compares two expressions. If one or both of the expressions is true, the result is true (1). If neither expression is true, the result is false (0).

EXOR Operator

numeric expression EXOR numeric expression

EXOR (exclusive or) compares two expressions. If only one of the expressions is true, the result is true (1). If both are true, or both are false, the result is false (0).

NOT Operator

NOT numeric expression

NOT returns the opposite of the logical value of an expression. If the expression is true (non-zero), the result is false. If the expression is false (zero), the result is true (1).

Examples

Here are some examples of logical operations. For these examples, assume A=0, B=2, C=4, and D=4.

A<B AND C=D	True. Both relational expression A<B and C=D are true.
A AND C=D	False. The arithmetic value of A equals 0 (false).
A OR B	True. The arithmetic value of B is not 0 (so B is true).
A EXOR B	True. One value is true and one value is false.
NOT A	True. A is 0 (false).
NOT B OR NOT C	False. NOT B is false and NOT C is false.

Here's a truth table summarizing logical operations –

A	B	A AND B	A OR B	A EXOR B	NOT A	NOT B
T	T	1	1	0	0	0
T	F	0	1	1	0	1
F	T	0	1	1	1	0
F	F	0	0	0	1	1

DIV Operator

The DIV (integer division) operator returns the integer portion of the quotient. DIV is useful for rounding, extracting multiples of a number and checking orders of magnitude.

Example

For example, if a test needs to run for 451 minutes, DIV can be used to find the number of whole hours the test will run –

```
451 DIV 60 [EXEC]          7
```

The following formula illustrates how DIV is calculated –

```
A DIV B = SGN(A/B)*INT(ABS(A/B) )[1]
```

MOD Operator

The MOD (modulo) operator returns the integer remainder resulting from a division. It is useful for testing divisibility, grouping serial data, and print control (print every Nth calculation, for example). Given two values X and Y, X MOD Y is equal to X – (Y * INT (X/Y)).

Example

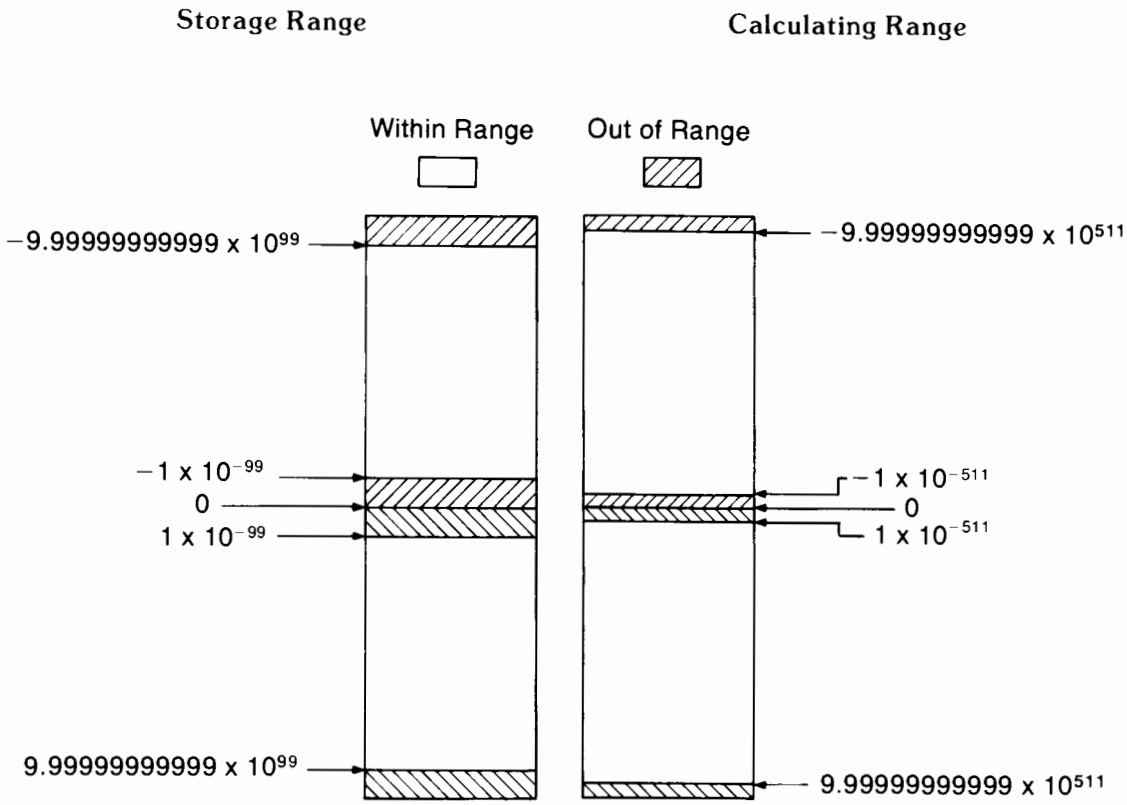
Referring to the 451-minute test in the previous example, MOD can be used to find the minutes remaining when 7 hours have passed –

```
451 MOD 60 [EXEC]          31
```

1 SGN (sign) and INT (integer) are functions which are covered later is this chapter.

Range

The range of values which can be entered, stored, or output on your computer is $-9.99999999999 \times 10^{99}$ through $-1 \times {}^{-99}$, 0, and 1×10^{-99} through $9.99999999999 \times 19^{99}$. However, the range of numbers the computer can operate on during intermediate calculations is $-9.99999999999 \times 10^{511}$, 0, and 1×10^{-511} through $9.99999999999 \times 10^{511}$.



Number Formats

Three formats are available to you for displaying and printing numbers: **standard**, **fixed point**, and **floating point** (scientific notation). No matter what the format, all numbers are output with a trailing blank and a leading blank or minus sign. It is always a good idea to set the desired format at the beginning of a program to avoid unexpected results from a previously set format.

The STANDARD Statement

The standard format is convenient for most computations since results appear in an easy-to-read form. Standard format is set at power on, reset, SCRATCH A, and when the STANDARD statement is executed –

```
STANDARD
```

In standard format, all significant digits of a number are output up to a maximum of twelve. Excess zeros to the right of the decimal point are suppressed; for example, 32.100000 would be output as 32.1. Leading zeros are truncated; for example 00223 is output 223.

In standard format, all numbers whose absolute values are between 1 and 10^{12} are output in fixed format showing all significant digits. Numbers between −1 and 1 are also output in fixed format if they can be represented precisely in twelve or fewer digits to the right of the decimal point. All other numbers are output in scientific notation as FLOAT 11.

The FIXED Statement

With fixed point, you specify the number of digits you want to appear to the right of the decimal point. For example, specifying two digits to the right of the decimal point would be useful for output of dollar and cent values. The FIXED statement sets fixed point format –

FIXED number of digits

The number-of-digits parameter is a numeric expression and is rounded to an integer to specify the number of digits to the right of the decimal point. Its range is 0 through 12.

Example

```
10   REM    This example shows the FIXED statement
20   FIXED 2                        ! 2 places after decimal
30   PRINT "8.7 IN FIXED 2:";8.7
40   FIXED 0                        ! No decimal point
50   PRINT "8.7 IN FIXED 0:";8.7    ! Number is rounded
60   END
```

```
8.7 IN FIXED 2: 8.70
8.7 IN FIXED 0: 9
```

Notice that the number is rounded to the specified format. Also notice that the decimal point is suppressed in FIXED 0.

When fixed point is set and the absolute value of the number is greater than or equal to 1E12 or would require more than 17 digits to represent it, the format temporarily reverts to floating point. For example, in FIXED 12, 100 000 is output as 1E+05, not 100000.000000000000.

The FLOAT Statement

When working with very large or very small numbers, the floating point format is most convenient. This format is set using the FLOAT statement –

FLOAT number of digits

The number-of-digits parameter is a numeric expression and is rounded to an integer to specify the number of digits to the right of the decimal point. Its range is 0 through 11.

A number output in floating point format follows this format –

- The leftmost non-zero digit of the number is the first digit output. If the number is negative, a minus sign precedes this digit; if the number is positive or zero, a space precedes this digit.
- A decimal point follows the first digit, except in FLOAT 0.
- The specified number of digits follows the decimal point.
- Then the character E appears followed by a plus sign or minus sign and a two-digit exponent, representing a positive or negative power of ten. The exponent specifies the power of 10 by which the mantissa should be multiplied in order to express the number in fixed point format.

Examples

Here are some numbers and how they are output in various modes –

```
10   REM      Examples of various output modes
20   PRINT "STANDARD","FIXED 4","FLOAT 3"
30   DATA 15,.054762,-.0006,2.75327,271,2.4
40   FOR I=1 TO 6
50      READ X                      ! READ a number
60      STANDARD                    ! Set STANDARD mode
70      PRINT X,
80      FIXED 4                     ! Set FIXED 4
90      PRINT X,
100     FLOAT 3                     ! Set FLOAT 3
110     PRINT X
120  NEXT I
130  END
```

```
STANDARD          FIXED 4          FLOAT 3
 15                15.0000          1.500E+01
 .054762           .0548            5.476E-02
-.0006            -.0006           -6.000E-04
 2.75327           2.7533           2.753E+00
 271               271.0000         2.710E+02
 2.4               2.4000           2.400E+00
```

Rounding

A number is rounded before being displayed or printed if there are more digits to the right of the decimal point than the number format allows. In either case, the rounding is performed as follows: The first excess digit on the right is checked. If its value is 5 or greater, the digit to the left is incremented (rounded up) by one; otherwise it is unchanged. In either case, the number remains unchanged internally.

Example

```
10   !    Output format doesn't change number internally
20   A=1.235                        ! 3 decimal places
30   FIXED 2                        ! Set FIXED 2
40   PRINT "A IN FIXED 2:";A        ! A is rounded
50   FIXED 3                        ! Set FIXED 3
60   PRINT "A IN FIXED 3:";A        ! Unchanged internally
70   END
```

```
A IN FIXED 2: 1.24
A IN FIXED 3: 1.235
```

Significant Digits

Significant digits are those which determine the internal accuracy with which a numeric variable is represented. The number format for output has no effect on this.

Math Functions and Statements

Math functions available on your computer are explained in this section. Parentheses must enclose the numeric expression used as the argument of the function if it contains any operators. For example, SINA+B does not equal SIN(A+B), but equals (SIN(A))+B. Parentheses enclose the expression when listed. Examples of two functions are combined in some cases.

General Functions

ABS numeric expression — Returns the absolute value of the expression.

```
10   REM  This example shows ABS function
20   INPUT "PRINTER SELECT CODE?",A
30   IF SGN(A)=1 THEN 60
40   PRINT "SELECT CODE CAN'T BE NEGATIVE"
50   PRINT "ABSOLUTE VALUE- ";ABS(A);" -WILL BE USED"
60   PRINTER IS ABS(A)
70   END
```

```
SELECT CODE CAN'T BE NEGATIVE
ABSOLUTE VALUE-  6  -WILL BE USED
```

DROUND (numeric expression, number of significant digits) — The digit round function returns the numeric expression rounded to the specified number of significant digits. The number-of-significant-digits parameter is rounded to an integer. If the specified number of digits is greater than 12, no rounding takes place. If it is less than 1, 0 is returned. DROUND is useful for checking equality to a specified number of digits, or standardizing internal value accuracy.

```
10   REM  This example shows DROUND function
20   A=12345
30   B=12346
40   IF DROUND(A,4)=DROUND(B,4) THEN GOSUB 80
50   PRINT "A=";A,"DROUND (A,4)=";DROUND(A,4)
60   PRINT "B=";B,"DROUND (B,4)=";DROUND(B,4)
70   STOP
80   PRINT "A AND B ARE EQUAL TO 4 SIGNIFICANT DIGITS"
90   RETURN
100  END
```

```
A AND B ARE EQUAL TO 4 SIGNIFICANT DIGITS
A= 12345            DROUND (A,4)= 12350
B= 12346            DROUND (B,4)= 12350
```

FRACT numeric expression — Returns the fractional part of the evaluated expression. (It is defined by this formula: expression − INT expression.)

INT numeric expression — The integer function returns the greatest integer which is less than or equal to the evaluated expression.

```
10   REM  This program shows FRACT and INT
20   A=22.987
30   B=-6.257
40   PRINT "VALUE";TAB(12);"FRACT";TAB(24);"INT"
50   PRINT A;TAB(12);FRACT(A);TAB(24);INT(A)
60   PRINT B;TAB(12);FRACT(B);TAB(24);INT(B)
70   END
```

```
VALUE        FRACT        INT
 22.987       .987         22
-6.257        .743         -7
```

MAX (list of numeric expressions) — The maximum function returns the greatest value in the list.

MIN (list of numeric expressions) — The minimum function returns the smallest value in the list.

```
10   REM  This program shows MAX and MIN
20   INPUT "FOUR VALUES",A,B,C,D
30   PRINT "FOUR VALUES:";A;B;C;D
40   PRINT "MAXIMUM:";MAX(A,B,C,D)
50   PRINT "MINIMUM:";MIN(A,B,C,D)
60   END
```

```
FOUR VALUES: 1  5  8  9
MAXIMUM: 9
MINIMUM: 1
```

PI

Returns an approximate value of π. It is represented internally as 3.1415926536.

```
10   REM   This program shows PI
20   PRINT "PI=";PI
30   DEG
40   INPUT "ANGLE IN DEGREES?",A
50   PRINT A;" DEGREES =";
60   A=A*PI/180     ! CONVERT DEGREES TO RADIANS
70   PRINT A;" RADIANS"
80   END
```

```
PI= 3.1415926536
 45  DEGREES = .7853981634  RADIANS
```

PROUND (numeric expression, power-of-ten position)

The power-of-ten round function returns the numeric expression rounded to the specified power-of-ten position. Specifying −2 is useful for output of money values.

4

```
1    REM    This program shows PROUND function
10   A=127.455
11   PRINT "ORIGINAL NUMBER:",A
20   FOR I=-2 TO 3
30      PRINT "POWER OF TEN:";I,PROUND(A,I)
40   NEXT I
50   END
```

```
ORIGINAL NUMBER:      127.455
POWER OF TEN:-2       127.46
POWER OF TEN:-1       127.5
POWER OF TEN: 0       127
POWER OF TEN: 1       130
POWER OF TEN: 2       100
POWER OF TEN: 3       0
```

RES

Returns the result of the last numeric computation which was executed from the keyboard.

RND

RANDOMIZE [numeric expression]

The random number function returns a pseudo random number greater than or equal to 0 and less than 1. The random number is based on a seed set to $\pi/180$ at power on, reset, SCRATCH A, and RUN. Each succeeding use of RND returns a random number which uses the previous one as a seed.

The seed can be modified by executing the RANDOMIZE statement. If the value of the expression is an integer, the value of the seed is set to 0 causing RND to return 0 each time it is used. To obtain a good seed, the expression should have as many digits to the right of decimal point as possible. A 1, 3, 7 or 9 is the most effective final digit. If no expression is specified, the computer arbitrarily resets the seed to one of 116 possible points; this is a more random selection.

```
10   REM  Example showing RND and RANDOMIZE
20   PRINT RND       ! Always set to same seed at RUN
30   PRINT RND*5
40   RANDOMIZE       ! Scrambles seed
50   PRINT RND
60   END
```

```
 .678219009345
 1.91093429525
 .152190093254
```

SGN numeric expression

The sign function returns a 1 if the expression is positive, 0 if it is 0 and −1 if it is negative.

```
10   REM  This program shows SGN
20   INPUT "NUMBER OF WORKERS",W
30   IF SGN(W)=-1 THEN 60          ! Branch when negative
                                     checking for proper value
40   PRINT "NUMBER OF WORKERS:",W
50   STOP
60   PRINT "CAN'T HAVE LESS THAN 0 WORKERS, ENTER AGAIN"
70   GOTO 20
80   END
```

```
NUMBER OF WORKERS:   1256
```

SQR numeric expression

The square root function returns the square root of a non-negative expression.

```
10    REM   Using SQR to find 3rd side of triangle
20    Side1=3
30    Side2=4
40    Hypotenuse=SQR(Side1^2+Side2^2)
50    PRINT "SIDE 1:";Side1
60    PRINT "SIDE 2:";Side2
70    PRINT "LENGTH OF HYPOTENUSE:";Hypotenuse
80    END
```

```
SIDE 1: 3
SIDE 2: 4
LENGTH OF HYPOTENUSE: 5
```

Logarithmic and Exponential Functions

EXP numeric expression	The exponential function returns the value of the constant Napierian e (= 2.71828182846 to twelve place accuracy) raised to the power of the computed expression.
LGT numeric expression	The common log function returns the logarithm (base 10) of a positive-valued expression.
LOG numeric expression	The natural log function returns the logarithm (base e) of a positive-valued expression.

```
 10   REM  This program shows EXP, LGT, and LOG
 20   A=1
 30   B=7
 40   PRINT "NUMBER";TAB(10),"EXP";TAB(27),"LGT";TAB(44),"LOG"
● 50   PRINT A;TAB(10),EXP(A);TAB(27),LGT(A);TAB(44),LOG(A)
● 60   PRINT B;TAB(10),EXP(B);TAB(27),LGT(B);TAB(44),LOG(B)
 70   END
```

```
NUMBER    EXP                LGT                LOG
 1         2.71828182844      0                  0
 7         1096.63315844      .845098040013      1.94591014905
```

Trigonometric Functions and Statements

The trigonometric functions use the angular unit mode: degrees, radians, or grads, which is currently set. A trigonometric statement is used to set the angular unit mode.

Radian mode is automatically set at power on, or when SCRATCH A, RUN, or reset is executed or when a subprogram is entered.

Degree Mode

To set degree mode, execute –

```
DEG
```

A degree is 1/360th of a circle.

Grad Mode

To set grad mode, execute –

```
GRAD
```

A grad is 1/400th of a circle and is commonly used in Europe.

Radian Mode

To reset radian mode, execute –

```
RAD
```

There are 2π radians in a circle.

Functions

ACS numeric expression	Returns the principal value of the arccosine of the expression in current angular units. The expression must be in the range −1 through +1.
ASN numeric expression	Returns the principal value of the arcsine of the expression in current angular units. The expression must be in the range −1 through +1.
ATN numeric expression	Returns the principal value of the arctangent of the expression in current angular units.
COS numeric expression	Returns the cosine of the angle represented by the expression in current angular units.
SIN numeric expression	Returns the sine of the angle represented by the expression in current angular units.
TAN numeric expression	Returns the tangent of the angle represented by the expression in current angular units.

```
10   REM  This program shows trig functions and statements
20   PRINT "ANGLE: 60"
30   FIXED 4
40   PRINT SPA(22);"SIN";SPA(7),"COS";SPA(7),"TAN";SPA(7),"ASN"
50   FOR I=1 TO 3
60       ON I GOSUB Degrees,Radians,Grads
70       PRINT SIN(60);TAB(30),COS(60);TAB(40),TAN(60);TAB(50);
80       S=SIN(60)
90       PRINT ASN(S)
100  NEXT I
110  STOP
120 Degrees:  DEG          ! Set DEGree mode
130  PRINT "DEGREES:",
140  RETURN
150 Radians:  RAD          ! Set RADian mode
160  PRINT "RADIANS:",
170  RETURN
180 Grads:  GRAD           ! Set GRAD mode
190  PRINT "GRADS:",
200  RETURN
210  END
```

```
ANGLE: 60
                        SIN        COS        TAN        ASN
DEGREES:               .8660      .5000     1.7321     60.0000
RADIANS:              -.3048     -.9524      .3200     -.3097
GRADS:                 .8090      .5878     1.3764     60.0000
```

Math Hierarchy

The order of execution for all mathematical operations is shown here.

When an expression has more than one operation, the order in which the computer performs the operations depends on the following hierarchy –

Highest Priority	( ) parentheses
↓	Functions
	^ (exponentiation)
	NOT, – (unary minus)
	*, /, MOD, DIV
	+, –
	All relational operators (=, <, >, >=, <=, < > or #)
	AND
Lowest Priority	OR, EXOR

An expression is scanned from left to right. Each operator is compared to the operator on its right. If the operator to the right has a higher priority, then that operator is compared to the next operator on its right. This continues until an operator of equal or lower priority is encountered: the highest priority operation, or the first of the two equal operations, is performed. Then any lower priority operations on the left are compared to the next operator to the right. This comparison continues until the entire expression is evaluated.

Parentheses

Parentheses allow lower priority operations to be performed before higher priority operations. When parentheses are used, they take highest priority. When parentheses are nested, like `(5*(4-2))`, the innermost quantity `(4-2)` is evaluated first.

Example

Here's the order of execution in solving an expression –

```
2+3*6/(7-4)^2
```

`2+3*6/(7-4)^2`	multiplication
`2+18/(7-4)^2`	evaluate parentheses
`2+18/3^2`	exponentiation
`2+18/9`	division
`2+2`	addition
`4`	result

Whenever you are in doubt as to the order of execution for any expression, use parentheses to indicate the order.

Using parentheses for "implied" multiplication is not allowed. So 4(5−2) must appear as `4*(5-2)`. The operator, `*`, must be used to specify explicit multiplication.

Math Errors-Recovery

Many math errors occur due to an improper argument or overflow; if a program is running, execution halts. It is possible to make some of these errors non-fatal so that execution doesn't halt by providing a default value for the number which is out of range. Using default values may alter the results of computations; be aware of this when using them.

The DEFAULT ON Statement

The default values are enabled by executing the DEFAULT ON statement –

```
DEFAULT ON
```

The errors and default values are –

Error (Number)	Default Value
Integer-precision overflow (20)	32767 or −32768
Short-precision overflow (21)	+ or − 9.99999E63
Full-precision overflow (22)	+ or − 9.99999999999E99
Intermediate result overflow (23)	+ or −9.99999999999E511[1]
TAN(N * PI/2), N:odd integer (24)	9.99999999999E511[1]
Zero to negative power (26)	9.99999999999E511[1]
LGT or LOG of zero (29)	−9.99999999999E511[1]
Division by zero (31)	+ or − 9.99999999999E511[1]
X MOD Y , Y=0 (31)	0

1 These values are used for intermediate results only. 99 is the greatest exponent for final results.

The DEFAULT OFF Statement

Default values are disabled by executing the DEFAULT OFF statement –

```
DEFAULT OFF
```

DEFAULT OFF is set at power on, reset, SCRATCH A, SCRATCH P, SCRATCH C and SCRATCH V.

Example

Here is an example that uses DEFAULT ON –

```
 10   REM     This program uses DEFAULT ON
● 20   DEFAULT ON                  ! Prevents error when I=0
 21   PRINT " I";SPA(5),"    10/I"
 30   FOR I=-5 TO 5
 40      PRINT I;SPA(5),10/I
 50   NEXT I
 60   END
```

```
 I          10/I
-5         -2
-4         -2.5
-3         -3.33333333333
-2         -5
-1         -10
 0          9.99999999999E+99
 1          10
 2          5
 3          3.33333333333
 4          2.5
 5          2
```

Try changing line 20 to DEFAULT OFF and running the program again.

4

Computer Museum

Chapter 5

Using Variables

A variable is a location in the memory of your computer that is assigned a value and accessed with a name. Algebraic formulas and other operations usually contain variables. The formula for the area of a circle, $A=\pi R^2$, uses two variables, A and R. You solve for A by assigning a value to R.

There are three types of numeric variables on your computer: full (12-digit), short (6-digit) and integer precisions. There is a fourth type of variable known as a string which is used to manipulate characters. Chapter 7 is devoted to strings.

page 56 • **LET** (assigns a value to one or more variables)
page 60 • **OPTION BASE** (declares 1 rather than 0 as the lower bound of array dimensions)
page 60 • **DIM** (dimensions and reserves memory for full-precision arrays and for strings)
page 61 • **INTEGER** (dimensions and reserves memory for integer-precision variables – simple and array)
page 62 • **SHORT** (dimensions and reserves memory for short-precision variables – simple and array)
page 62 • **REAL** (dimensions and reserves memory for real-precision variable – simple and array)
page 63 • **COM** (reserves space in a common memory area for numeric and string variables)
page 64 • **REDIM** (redimensions arrays)
page 66 • **DATA** (supplies values that are assigned by READ and MAT READ)
page 66 • **READ, MAT READ** (assigns values from a DATA statement to variables)
page 68 • **RESTORE** (repositions the DATA pointer)
page 71 • **INPUT, MAT INPUT** (lets values be assigned to variables from the keyboard)

Terms

- Name – a capital letter followed by 0 through 14 lowercase letters, numbers, or underscore characters.
- Array – a collection of up to 32 767 data items of the same type, with 1 to 6 dimensions.
- Dimensioning – defining the number of dimensions, elements per dimension and type of an array.
- Maximum size – the number of elements in an array when it is dimensioned.
- Working size – the total amount of elements being used currently.
- Array element – a single item of an array.
- Array identifier – (*) following the array name, used to specify all elements of the array collectively in an input or output operation.
- Subscripts – integers separated by commas and enclosed in parentheses for accessing an array element or dimensioning an array.
- Implicit definition – using an array element without dimensioning the array dimensions the array implicitly with 10 as the upper bound of each dimension.
- Redim subscripts – used to redimension an array and can be a numeric expression.
- DATA pointer – an internal mechanism that indicates which value in a DATA statement is accessed next.

Types

There are three types of numeric variables available with your computer.

- Full-precision (real) variables are represented internally with twelve significant digits and an exponent in the range −99 through 99. Full-precision variables don't need to be declared, but the REAL statement can be used for documentation purposes.
- Short-precision variables are represented internally with six significant digits and an exponent in the range −63 through 63. A short-precision number is declared in a SHORT or COM statement.
- Integer-precision variables have no digits following the decimal point. The range of integer-precision numbers is −32 768 through 32 767. An integer is declared in an INTEGER or COM statement.

All numbers are full precision unless otherwise specified using a SHORT, INTEGER or COM statement. Any excess digits input for a number are truncated when the number is stored in memory. For example, if you input 12345678912365 for a full-precision number, it is represented internally with 123456789123 in the mantissa.

Short and integer-precision variables are useful for conserving memory. All calculations are performed with full-precision accuracy, so short and integer precision numbers are converted before and after an operation, which can cause operations to take more time.

Forms

There are two forms that any type of variable may have.

- Simple (Nonsubscripted)
- Array – a collection of single data items.

Names

All variables must have a name. The name can be useful for describing what the variable is used for. Names must follow these rules –

- A name has between 1 and 15 characters.
- The first character must be a capital letter.
- The remaining characters must be lowercase letters, digits, or the underscore character obtained by pressing [SHIFT] [–].
- String names must be followed by $ (dollar sign).

Examples

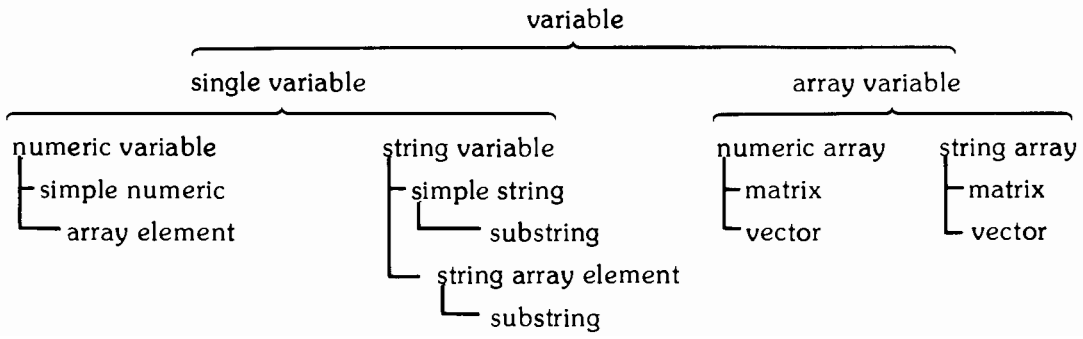
Here are some examples of variable names –

```
X
Social_security
Acct_number
Amount_owed
Payment
Interest
```

Any name can be used simultaneously for a simple numeric, simple string, numeric array and string array.

Variable Breakdown

Variables can be classified into various categories and subsets of the categories as shown in the diagram below. For example, any reference to a single numeric variable includes simple numerics and elements of numeric arrays. –



Using Variables at the Keyboard

Variables can be assigned values using an equals sign to create an **assignment statement**. For example, to assign 150 to Owed and 25 to payment, enter –

```
Owed=150 [EXEC]
Payment=25 [EXEC]
```

Now that some variables have assigned values, they can be used in place of numbers in math calculations –

```
Owed=Owed-Payment [EXEC]
Interest=Owed*.06 [EXEC]
```

The LET Statement

A simple numeric variable can be assigned a value in a program using the LET statement –

[LET] simple variable [= simple variable...] = numeric expression

Implied LET

5

Omitting LET is an implied LET or implied assignment.

Examples

```
10   X=12                  ! Assign 12 to X
20   LET M=X               ! Assign the value of X to M
30   LET Y=3*4             ! Assign 12 to Y--use any expression
40   Data1=32.17           ! Assign 32.17 to Data1
50   A=B=C=0               ! Assign 0 to A,B & C--multiple assign
60   Counter=Counter+1     ! Adds 1 to the value of Counter
70   STOP
```

If a numeric variable is used in a computation and hasn't been assigned a value, 0 is used as its value.

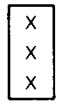
To check the current value of a variable, you can type in its name, then press [EXEC]. This can also be done while a program is running in live keyboard mode. You may get an unexpected result if a subprogram is currently executing and the variable isn't defined in the subprogram.

The values of variables are erased by executing SCRATCH V (except those in common), SCRATCH C, SCRATCH A, or SCRATCH P.

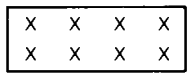
Array Variables

An array variable (array) is a collection of data items of the same type. An array can have one to six dimensions and up to 32 767 elements. It is a convenient tool for handling related data items within a program. Use arrays when you have related values and need to access any of them at any time, when you need to sort many values, when you need to keep track of a table of values or when you have interrelationships between data items (a person's age, height, weight, phone number, and Social Security number for example).

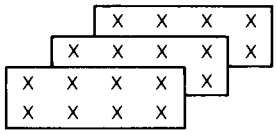
A one-dimensional array (also known as a **vector**[1]) can be thought of as a column of items. The following represents a vector having three items; X represents one item.



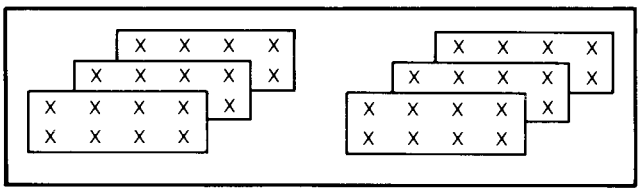
The structure of a two-dimensional array (also known as a **matrix**[1]) is rows and columns. Here is a representation of a 2 by 4 (2x4) array.



The structure of a three-dimensional array can be thought of as a series of two-dimensional arrays. Here is a representation of a 3x2x4 array. Your computer interprets it as three 2 by 4 arrays.



A four-dimensional array can be thought of is a series of three-dimensional arrays. Here is a representation of a 2x3x2x4 array.



A five-dimensional array is a series of four-dimensional arrays and a six-dimensional array as a series of five-dimensional arrays. Your arrays can be structured according to your needs.

1 Vectors and matrices are special types of arrays. Any reference in this manual to an array also includes matrices and vectors.

Defining the Size of an Array

To use an array, your computer must know its size. An array can be initially defined in a variable declarative statement (DIM, COM, REAL, SHORT, or INTEGER). There, its maximum size is indicated by specifying the number of dimensions and size of each dimension. This is known as **dimensioning** the array. An array is limited by memory size to no more than 32 767 items.

When an array is dimensioned, its physical or **maximum size** is defined. The **working size** of an array is the **total** amount of elements being used. A new working size can be specified in a REDIM statement or in certain array operation statements.

Implicit Definition

If an array element (discussed next) is used in a program or keyboard computation, but the array has not been defined in a variable declarative statement, the array is then **implicitly dimensioned**. This means that an array is dimensioned having the number of dimensions indicated by the array element. The upper bound of each dimension is 10; the lower bound is 0 or 1, depending on the current OPTION BASE setting. OPTION BASE is covered later in this chapter.

5

Array Elements

Each **element** in an array can be accessed by using subscripts, then used like a simple variable. An array element is a type of single variable. In a 2x4 array called M, M(1,2) refers to the element in array M which is in row 1, column 2. It can be assigned a value and used in calculations and other programming operations.

Example

```
10    OPTION BASE 1
20    DIM M(2,4)          !Dimensions array to 2 dimensions.
30    M(1,2)=10           !Assigns 10 to one element.
40    A=M(1,2)/7
50    PRINT M(1,2),A      !Outputs two values
60    END
```

Array Identifier

All elements of an array (in its working size) can be specified collectively in an input or output operation by using the array identifier: (*) after the name. For example –

```
PRINT M(*)
```

prints the entire array M.

Declaring and Dimensioning Variables

Five variable declarative statements – COM, DIM, INTEGER, SHORT and REAL – are used to dimension arrays and strings and declare the precision of numeric variables. These statements also reserve space in memory for the specified variables when the program is run.

Subscripts

When you dimension an array, you declare the number and size of dimensions with **subscripts**. The rightmost subscript represents the length of a row and varies fastest.

Subscripts are integers separated by commas and enclosed in parentheses. In addition, subscripts can be numeric expressions when used in a subprogram to dimension an array. The range of each subscript is −32 767 through 32 767, but the size of an array is limited to no more than 32 767 elements by memory size.

Subscripts can specify just the upper bound for each dimension. The lower bound for each dimension is then 0. The OPTION BASE statement can be used to change it to 1; OPTION BASE is covered next.

Subscripts can also be used to specify the lower as well as upper bound of each dimension. This is done by separating the upper and lower bounds with a colon. This allows you to have more meaningful indexes, specifying 1967-1978 rather than 1-12, for example.

Example

Here are some ways to dimension a 2x4 array –

```
10    DIM M(1,3)     ! 0 to 1, 0 to 3
20    DIM N(1:2,1:4)
30    DIM O(2:3,5:8)
40    STOP
```

The OPTION BASE Statement

When dimensioning arrays, you may want to specify that the default lower bound for dimensions be 1 rather than 0. This can be done using the OPTION BASE statement –

```
OPTION BASE 1
```

This statement must come **before** any of the variable declarative statements used in a program. Then, any lower bound not specified is 1.

If OPTION BASE 1 is not declared in a program, you may wish to include the statement –

```
OPTION BASE 0
```

for documentation purposes.

The OPTION BASE statement can't be executed from the keyboard.

The DIM Statement

The DIM (dimension) statement is used to dimension and reserve memory for full-precision numeric arrays and initialize each element to 0. It is also used to dimension and reserve storage space for simple strings and string arrays and initialize all strings to the null string.

DIM item [, item...]

The item can be –

numeric array (subscripts)
simple string [number of characters]
string array (subscripts) [[number of characters]]

Example

```
10   OPTION BASE 1      ! 1 is lower bound for array dimensions
20   DIM A(4,4)         ! 2-dimensional array with 16 elements
30   DIM Array(2,3,5),Times(5000)
40                      ! Line 30 dimensions a 3-dimensional and
                          a 1-dimensional array
50   STOP
```

Remember these things when using DIM –

- The maximum number of dimensions that can be specified is six. The range of subscripts is −32 767 to 32 767. No array can have more than 32 767 elements. No simple string can be longer than 32 767 characters. The size of arrays or strings may be limited by available memory, however.
- The maximum length of a string (number of characters) can be specified with any numeric expression except one containing a multiple-line user-defined function.
- The DIM statement must be executed via a program, not from the keyboard. Its location in a program is arbitrary, though it must be after any OPTION BASE statement. At pre-run initialization, the variables are dimensioned and initialized.
- DIM need not be used to assign space for strings with 18 characters or less or for arrays having upper bounds of 10 or less. These can be dimensioned implicitly. This, however, may use memory inefficiently by creating arrays or strings which are larger than needed.
- A program can have more than one DIM statement, but the same variable name can be declared only once in a program segment. Therefore, arrays of differing dimensions can't have the same name.

The INTEGER Statement

The INTEGER statement is used to dimension and reserve memory for integer precision variables – simple and array. Integer-precision variables can be used to conserve memory; all calculations are performed with full-precision accuracy however, so a conversion is made before and after an operation, which takes more time.

INTEGER numeric variable$_1$ [⟨subscripts⟩] [, numeric variable$_2$ [⟨subscripts⟩], ...]

Example

```
30  OPTION BASE 1
40  INTEGER X,Y(2,2)
```

declares X to be an integer and Y to be an integer array of four elements.

The SHORT Statement

The SHORT statement is used to dimension and reserve storage for short-precision variables – simple and array. Short-precision variables can be used to save memory. All calculations are performed with full-precision accuracy, however, so a conversion is made before and after an operation, which takes more time.

SHORT numeric variable$_1$ [(subscripts)] [, numeric variable$_2$ [(subscripts)], ...]

Example

```
30 SHORT A(4,4),B(3,3,3),D
```

declares A and B as short-precision arrays and D as a simple, short-precision variable.

The REAL Statement

The REAL statement is used to dimension and reserve memory for full-precision (real) variables – simple and array.

REAL numeric variable$_1$ [(subscripts)] [, numeric variable$_2$ [(subscripts)], ...]

5

Example

```
10   REAL M(2,2,2,2),N
```

dimensions the array M and declares the simple variable N.

Since the DIM statement can also be used to dimension full-precison variables, the REAL statement can be used for documentation purposes to specify which variables are full precision.

The COM Statement

The COM statement is used to dimension and reserve memory for simple and array variables. This includes strings and all three numeric precisions. COM is unique because it reserves memory space in a special "common" area which allows data to be transferred to and from subprograms or to other programs when each program or subprogram has corresponding COM statements.

COM item [, item, ...]

The item can be –

simple numeric
numeric array (subscripts)
simple string [[number of characters]]
string array (subscripts) [[number of characters]]

In addition, any one of the type-words – INTEGER, SHORT, REAL – can precede one or more numeric variables. All variables following a numeric type word have that precision until another type is specified or a string is declared.

Example

```
10   OPTION BASE 1
20   COM A,B(2,4),C$,D,INTEGER E
30   COM F$(5)[24],G,SHORT H(5),J
```

The variables A,B(2,4), D and G are all full precision. Full precision is assumed at the beginning of the COM list and for numeric variables which are declared after any string. Since all variables following a numeric type word have that precision until another type is specified or a string is declared, both H(5) and J are short precision.

The items declared in corresponding COM statements in separate programs and subprograms must correspond to preserve values. Each item must be of the same type – integer, short, full-precision and string – as the corresponding item in other COM statements. Arrays must have the same upper and lower bounds for each dimension. Strings must have the same number of characters dimensioned. Names need not match, however.

COM statements in separate programs need not have the same number of items. A shorter COM statement in a succeeding main program causes the extra data from the first COM statement to be lost. A longer COM list in a succeeding program causes the new elements of the second COM statement to be initialized to 0 or the null string.

Redimensioning an Array

Redimensioning an array allows you to reorganize it into a more useful configuration. When an array is redimensioned, it is given a new working size. Any elements not included in the new working size are ignored, but are still part of the array. Thus, when new values are assigned to elements of a redimensioned array, the values of the unused elements are not changed.

A redimensioned array must retain the same number of dimensions as orginally specified. Also, the total number of elements can't exceed the number originally specified.

Redimensioning of an array can be explicitly specified in many of the array statements. MAT INPUT and MAT... IDN are two examples.

Redimensioning can also be implicitly specified in many of the array operation statements. For example, adding the elements of two 3x3 arrays and storing the sums in a 5x5 array causes the result array to be redimensioned to 3 x 3.

The REDIM Statement

A new working size for an array can be established by using the REDIM statement.

5

REDIM array variable (redim subscripts) [, array variable (redim subscripts), ...]

Redim subscripts have the same characteristics as dimensioning subscripts. In addition, they can be any numeric expression, not just an integer. If you redimension a string array, you can't change the string (element) length.

Examples

Here are some example REDIM statements –

```
  10   REM  These are example REDIM statements
  20   DIM A(5,5),B(2,5,6),C(10)
• 30   REDIM B(1,4,-3:1)  ! Upper & lower bounds can be specified
  40   X=2
  50   Y=3
• 60   REDIM A(X,Y),C(7)  ! Subscripts can be numeric expressions
  70   END
```

The following program illustrates what happens when an array is redimensioned –

```
10   OPTION BASE 1
20   DIM A(3,5)            ! A 3 by 5 array
30   MAT A=(7)             ! All elements equal 7
40   PRINT "ARRAY A--3 BY 5",A(*);
50   REDIM A(2,2)          ! Redimension to 2 by 2
60   MAT A=(1)             ! Elements in new working size =1
70   PRINT "ARRAY A--NOW A 2 BY 2";A(*);
80   REDIM A(3,5)          ! Back to original size
90   PRINT "ARRAY A--BACK TO A 3 BY 5"
91   PRINT "NOTE WHICH ELEMENTS CHANGED"
100  MAT PRINT A;
110  END
```

```
ARRAY A--3 BY 5
 7  7  7  7  7

 7  7  7  7  7

 7  7  7  7  7


ARRAY A--NOW A 2 BY 2
 1  1

 1  1


ARRAY A--BACK TO A 3 BY 5
NOTE WHICH ELEMENTS CHANGED
 1  1  1  1  7

 7  7  7  7  7

 7  7  7  7  7
```

5

More Ways to Assign Values to Variables

Values can be assigned to variables during a program, either from within the program or input directly from the keyboard. Besides LET, you can use –

```
READ, MAT READ, DATA, RESTORE
INPUT, MAT INPUT
```

The READ and DATA Statements

To assign values to variables from within a program, the DATA statement is used with READ. READ and DATA allow you to store values in a program. The DATA statement(s) provides values that are assigned to the variables. The READ statement specifies the variables for which values are to be obtained from DATA statements. READ and DATA are programmable only; they can't be executed from the keyboard.

DATA constant or text [, constant or text, ...]
READ variable name [, variable name, ...]

In the DATA statement, text for strings can be quoted or unquoted. A constant could thus be interpreted as either a numeric value or as unquoted text. The location of the DATA statements within a program segment is unimportant. If there are multiple DATA statements, make sure they are in the order you want, since the program accesses them in order.

The variables specified in the READ statement can be any single variable or an array identifier (*) following the array name.

Example

```
10   !   This program shows READ and DATA statements
20   DIM Totals(1:3)
30   DATA 88,77,April,"100",95,78,100
40   READ A,A$,Month$,Miles$,Totals(*)
50   PRINT "A=";A,"A$=";A$
60   PRINT "Month$=";Month$,"Miles$=";Miles$
70   PRINT "Totals(*)=";Totals(*);
80   END
```

```
A= 88                A$=77
Month$=April         Miles$=100
Totals(*)=
 95  78  100
```

Notice that an unquoted value in the DATA statement (77) can correspond to a string variable in the READ statement (A$). It is interpreted as unquoted text in this case.

The MAT READ Statement

The MAT READ statement specifies entire arrays for which values are to be read from DATA statements.

MAT READ array variable [(redim subscripts)] [, array variable [(redim subscripts)], ...]

The working size of the array can be altered by including the redim subscripts. When redimensioning, the total number of elements can't be greater than the number originally dimensioned. The number of dimensions can't change. The subscripts can be any numeric expression except one containing a multiple-line user-defined function (FN) reference.

The MAT READ statement is programmable only; it can't be executed from the keyboard.

Array elements are read in order with the rightmost subscript varying fastest.

Example

```
10   OPTION BASE 1
20   DIM A(5,5,5)
30   DATA 1,2,3,4,5,6,7,8,9
40   MAT READ A(2,2,2)    ! Redimension A
50   MAT PRINT A;
60   END
```

```
 1  2

 3  4

 5  6

 7  8
```

Values are read in the following order –

A(1,1,1), A(1,1,2), A(1,2,1), A(1,2,2), A(2,1,1), A(2,1,2), A(2,2,1), A(2,2,2)

DATA Pointer

The computer uses an internal mechanism called a **DATA pointer** to locate the next data element that is to be read. The leftmost element of the lowest-numbered DATA statement in the current environment is read first. After this element is read and another value is required by READ, the DATA pointer repositions itself one element to the right, and continues to do so each time another data element is read. After the last element in a DATA statement is read and another value is required by READ, the DATA pointer locates the next higher numbered DATA statement and repositions itself at the first element in that statement. If there are no higher-numbered DATA statements, the data pointer remains at the end of the previous DATA statement; any effort to read additional data results in ERROR 36.

The RESTORE Statement

The DATA pointer can be repositioned to the beginning of any DATA statement in the same program segment, so that values can be reused, by using the RESTORE statement –

RESTORE [line identifier]

If no line identifier is specified, the pointer is repositioned to the beginning of the lowest-numbered DATA statement in the same program segment.

If the specified line is not a DATA statement, the first DATA statement following the specified line is accessed.

Examples

Here are some examples of DATA, READ, MAT READ and RESTORE.

This example shows that several READ statements can apply to the same DATA statement. It also shows that string values can be quoted or unquoted, though quotes are not part of the string; notice that 7.31 is a string value assigned to A$.

5

```
 10   DIM A(1:3)
● 20   READ A(*)          ! Reads the 4,5 and 6
● 30   READ A$,B1         ! Reads the 7.31 and the 2.69
● 40   READ X$            ! Reads "Hours"   ! Quotes not read
● 50   DATA 4,5,6,7.31,2.69,"Hours"
 51   PRINT A(*);
 52   PRINT A$,B1
 53   PRINT X$
 60   STOP
```

```
 4  5  6


7.31                    2.69
Hours
```

This example illustrates use of RESTORE and MAT READ. The values in line 30 are used as the values of five simple variables, then re-used as the values in array B.

```
  10   OPTION BASE 1
  20   DIM B(5)
• 30   DATA 10,12.7,3,15,8
  40   FOR I=1 TO 5
• 50      READ C
  60      PRINT "C=";C,"SQUARE ROOT OF C=";SQR(C)
  70   NEXT I
  80   PRINT
  90   PRINT            ! Lines 80 and 90 give blank lines
• 100  RESTORE 30       ! Lets the data in line 30 be re-used
• 110  MAT READ B
  120  PRINT "ARRAY B:",B(*);
  130  END
```

Here is the output –

```
C= 10                SQUARE ROOT OF C= 3.16227766016
C= 12.7              SQUARE ROOT OF C= 3.56370593624
C= 3                 SQUARE ROOT OF C= 1.73205080756
C= 15                SQUARE ROOT OF C= 3.8729833462
C= 8                 SQUARE ROOT OF C= 2.82842712474


ARRAY B:
 10  12.7  3  15  8
```

The INPUT Statement

The INPUT statement allows the user of a program to interact with the program and input varying data. Values in the form of expressions can be assigned to variables from the keyboard at the request of the program –

INPUT ["prompt",] variable name [, ["prompt",] variable name,...]

When the INPUT statement is executed, a ? or the prompt, if present, is displayed. The prompt is any combination of characters; it can be used to show the user for what variable a value is being requested. Each prompt applies only to the variable to its right. If no characters are present between the quotes, nothing is displayed. Any variable not preceded by a prompt uses a question mark by default. A value can then be input for each variable designated in the INPUT statement. Values are entered into the computer by pressing CONT or STEP.

Example

For instance, in the statement –

```
10 INPUT "Age and Employee Number", Age,Numb,"Pay",Pay
```

three values are requested.

Values can be assigned individually or separated by commas in groups. Values input for strings can be quoted or unquoted. Quotation marks can't be input as part of the string's value. An unquoted value for a string can't contain a comma or exclamation point and all leading and trailing blanks are deleted.

Example

For example, the values 24, 2592 and 350 can be assigned to the variables in the example above in many ways; here are two –

24 [CONT] 2592 [CONT] 350 [CONT]

or

24, 2592 [CONT] 350 [CONT]

In both cases, the ? or prompt reappears after [CONT] is pressed until all three values are input.

Pressing [CONT] without entering values when an input is requested causes execution to continue with the statement following the INPUT statement, even if additional values are still requested. [STEP] or the CONT command can also be used. Variables not assigned values retain their previous value.

Example

```
  40   X=5
  50   PRINT X
• 60   INPUT "INPUT VALUES FOR A, B AND X",A,B,X
  70   PRINT A,B,X
  80   END
```

If you respond to the INPUT statement with –

26, 19 [CONT] [CONT]

The output is as shown. Notice X retains the value 5.

```
5
26                    19                    5
```

The variable list can also include array identifiers.

Example

```
100  INPUT A,B(*)
```

requests values for the simple variable A and the array B.

The INPUT statement is programmable only; it can't be executed from the keyboard.

The MAT INPUT Statement

Entire arrays can be given values from the keyboard and optionally redimensioned using the MAT INPUT statement –

> MAT INPUT array variable [(redim subscripts)] [, array variable [(redim subscripts)], ...]

When MAT INPUT is executed, a ? appears in the display line. Values in the form of numeric or string expressions can be entered separately, or in groups. As with the INPUT statement, values are stored by pressing (CONT) or (STEP) or using the CONT command.

The first array with redim subscripts is always redimensioned. Redimensioning of any but the first array takes place only if a value is input for at least one element in the array.

Example

```
10   OPTION BASE 1
20   DIM Array(5,3,5),B(10),C(2,2)
30   MAT Array=(0)
40   MAT B=(9)
50   MAT INPUT Array(2,2,5),B(5),C   ! REDIM Array
60   MAT PRINT Array;B;C;
70   END
```

```
 1  2  3  4  5

 6  0  0  0  0

 0  0  0  0  0

 0  0  0  0  0


 9  9  9  9  9  9  9  9  9  9


 0  0

 0  0
```

When this is executed, 29 separate values are requested. If

1, 2, 3, 4, 5, 6 [CONT] [CONT]

are entered, the only values that would change are the elements of Array with subscripts (1,1,1), (1,1,2), (1,1,3), (1,1,4), (1,1,5), (1,2,1). Array B is not redimensioned because no elements were input for it.

The MAT INPUT statement can't be executed from the keyboard.

Storage of Variables

To determine how many bytes a variable requires when stored **in memory** (storage on a mass storage medium is different; see Chapter 11), use the following tables.

Simple Variable	Amount of Memory Used
Full precision	10 bytes
Short precision	6 bytes
Integer	4 bytes
String	6 bytes + length (1 byte per character, rounded up to an even integer)

Array Variable	Amount of Memory Used[1]
Full precision	10 bytes + 4 bytes per dimension + 8 bytes per element
Short precision	10 bytes + 4 bytes per dimension + 4 bytes per element
Integer	10 bytes + 4 bytes per dimension + 2 bytes per element
String	12 bytes + 4 bytes per dimension + 2 bytes per element + length of each string (1 byte per character, rounded up to an even integer)

1 See the "Memory" section of the Reference Tables.

5

Chapter 6
Array Operations

Computer Museum

page 76 • **MAT...CON** (assigns 1 to every element in an array)
page 76 • **MAT...ZER** (assigns 0 to every element in an array)
page 77 • **MAT-Initialize** (assigns a constant value to every element in an array)
page 78 • **MAT-Copy** (copies all the values of one array into another)
page 79 • **MAT-Scalar operation** (performs an operation on every element in an array with a constant scalar)
page 80 • **MAT-arithmetic operation** (performs an operation on corresponding elements of two arrays)
page 81 • **MAT-function** (operates on every element in an array with a system function)
page 82 • **MAT...IDN** (establishes an identity matrix)
page 83 • **MAT-matrix multiplication** (performs multiplication on two matrices)
page 85 • **MAT...INV** (finds the inverse of a matrix)
page 87 • **MAT...TRN** (finds the transpose of a matrix)
page 87 • **MAT...CSUM** (finds all the column sums of a matrix)
page 88 • **MAT...RSUM** (finds all the row sums of a matrix)

Functions

SUM (sum of all the elements)
ROW (number of rows)
COL (number of columns)
DOT (dot product)
DET (determinant)

Terms

- Identity matrix – a square matrix with all elements equal to 0 except the main diagonal, which all equal 1.
- Scalar – a numeric expression used as a constant in mathematic operations.

Assigning a Constant Value

Three statements allow a constant value to be assigned to every element in a previously dimensioned array.

1. MAT...CON

The MAT...CON statement assigns the value 1 to every element –

MAT array variable = CON [(redim subscripts)]

When executed, all elements in the current size of the array are assigned the value 1. The current size can also be redimensioned by including the redim subscripts. The redimensioning is done before the assignment takes place.

Example

In this example the value 1 is assigned to 9 elements of the array A –

```
10   OPTION BASE 1
20   DIM A(15,15)
30   MAT A=CON(3,3)   ! Assigns 1 to 9 elements of A
40   MAT PRINT A;
50   END
```

```
 1  1  1

 1  1  1

 1  1  1
```

6

2. MAT...ZER

The MAT...ZER statement sets all elements in a numeric array to 0. You can also redimension the array.

MAT array variable = ZER [(redim subscripts)]

Again, the optional redimensioning takes place before the assignment.

Example

```
10   REM   This example shows MAT...ZER
20   OPTION BASE 1
30   DIM X(20)
40   MAT X=ZER(15)      ! Redimension array X also
50   MAT PRINT X;
60   END
```

```
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

15 elements of the array X are assigned the value 0.

3. MAT-Initialize

The MAT – **Initialize** statement assigns the value of the numeric expression to every element in a numeric array.

MAT array variable = (numeric expression)

Example

```
10    !   This example shows initialization of array
20    OPTION BASE 1
30    DIM Areas(2,2,2)
40    MAT Areas=(2*PI)
50    FIXED 3
60    PRINT Areas(*);
70    END
```

```
 6.283  6.283

 6.283  6.283

 6.283  6.283

 6.283  6.283
```

This assigns the value of 2*PI to every element in Areas.

The numeric expression is evaluated once; it is converted to the numeric type (INTEGER, SHORT, REAL) of the array, if necessary.

Example

```
10    INTEGER X(4,4)     ! OPTION BASE value is 0
20    MAT X=(PI)         ! Value of PI rounded to an integer
30    MAT PRINT X;
40    END
```

```
 3  3  3  3  3

 3  3  3  3  3

 3  3  3  3  3

 3  3  3  3  3

 3  3  3  3  3
```

Line 20 causes the value 3 to be assigned to every element in X.

The MAT-Copy Statement

The MAT – **Copy** statement copies the value of each element of a numeric array into the corresponding element of the result array.

MAT result array = operand array

The two arrays must have the same number of dimensions. The number of elements in the result array must be greater than or equal to the number of elements in the current size of the operand array.

Example

```
10   OPTION BASE 1
20   DIM C(4,4),D(2,2)
30   PRINT "MATRIX C:",C(*);
40   MAT D=(2.5)
50   PRINT "MATRIX D:",D(*)
60   MAT C=D          ! C is redimensioned to size of D
70   PRINT "MATRIX C:",C(*)
80   END
```

```
MATRIX C:
 0  0  0  0

 0  0  0  0

 0  0  0  0

 0  0  0  0


MATRIX D:
 2.5                 2.5

 2.5                 2.5


MATRIX C:
 2.5                 2.5

 2.5                 2.5
```

The working size of array C is redimensioned to be a 2 by 2 array, then the values of array D are copied into the elements of array C.

Mathematical Operations

There are various mathematical operations that can be performed with arrays. These are covered next.

Scalar Operations

The **scalar operation** statement allows an arithmetic or relational operation to be performed with each element of an array using a constant scalar (any numeric expression). The result of the operation becomes the value of the corresponding element of the result array. The operators that can be used are –

+	<
–	>
*	<=
/	>=
=	<> or #

MAT result array = operand array operator (scalar)
MAT result array = (scalar) operator operand array

Example

In this example, each element in array C is multiplied by 4 and the result is stored in the corresponding element of array B.

```
10   OPTION BASE 1
20   DIM B(3,3),C(2,2)
30   FOR I=1 TO 2        ! These loops assign values to C
40      FOR J=1 TO 2
50         C(I,J)=I+J
60      NEXT J
70   NEXT I
80   MAT B=C*(4)         ! Each element in C is multiplied by 4;
                         B is also redimensioned
90   PRINT "MATRIX C:",C(*)
100  PRINT "MATRIX B:",B(*)
110  END
```

```
MATRIX C:
 2                    3

 3                    4


MATRIX B:
 8                    12

 12                   16
```

The two arrays must have the same number of dimensions. The result array can't be smaller than the operand array. The array is redimensioned before the operation so that it has the same working size as the operand array.

Arithmetic Operations

The **arithmetic operation** statement allows an arithmetic or relational operation to be performed with corresponding elements of two numeric arrays; the result becomes the value of the corresponding element in the result array.

MAT result array = operand array operator operand array

Examples

In this example, corresponding elements of arrays A and B are added.

```
 10   OPTION BASE 1
 20   DIM A(2,2),B(2,2),Sum(3,3)
 30   FOR I=1 TO 2      ! Assigning values to A and B
 40      FOR J=1 TO 2
 50         A(I,J)=J+I
 60         B(I,J)=J*I
 70      NEXT J
 80   NEXT I
 100  MAT Sum=A+B       ! EACH ELEMENT OF Sum IS THE SUM OF
                        CORRESPONDING ELEMENTS OF A AND B
 110  PRINT "ARRAY A:",A(*),"ARRAY B:",B(*)
 120  PRINT "SUM OF A AND B:",Sum(*)
 130  END
```

6

```
ARRAY A:
 2                 3

 3                 4


ARRAY B:
 1                 2

 2                 4


SUM OF A AND B:
 3                 5

 5                 8
```

In this example, corresponding elements of arrays Hours and Rate are multiplied together and the result stored in array Pay.

```
10   OPTION BASE 1
20   DIM Pay(5),Hours(5),Rate(5)
30 Rates:   DATA  5.25,5.00,4.25,7.15,4.80
40 Hours:   DATA  40,42,28,39,40
50   MAT READ Rate,Hours
60   MAT Pay=Hours.Rate
70   REM  EACH ELEMENT OF Pay IS THE PRODUCT OF
          CORRESPONDING ELEMENTS OF Hours AND Rate
80   PRINT "PAY TOTALS:",Pay(*);
90   END
```

```
PAY TOTALS:
 210  210  119  278.85  192
```

The following operators are allowed –

+	–
. (multiply)	/
=	<> or #
>	<
>=	<=

Notice that multiplication is indicated by a period. An asterisk indicates matrix multiplication which is different and is covered later in this chapter.

The result and operand arrays must have the same number of dimensions. The operand arrays must have the same number of elements in each dimension; the result array can't be smaller.

Functions

The **function** statement allows each element in the operand array to be evaluated by the specified function. The result becomes the corresponding element of the result array.

MAT result array = function operand array

The function must be a single-argument system function like SIN, ABS or SQR.

Example

In this example, the square root of each element in array A is assigned to the corresponding element in array B.

```
10   OPTION BASE 1
20   DIM A(3,3),B(3,3)
30   DATA 25,13,55,66,48,15,36,64,80
40   MAT READ A
50   MAT B=SQR(A) ! Each element of array B is square root of
                    corresponding element of array A
60   PRINT "MATRIX A:",A(*)
70   FIXED 5
80   PRINT "MATRIX B - THE SQUARE ROOTS:",B(*)
90   END
```

```
MATRIX A:
 25                    13                    55

 66                    48                    15

 36                    64                    80


MATRIX B - THE SQUARE ROOTS:
 5.00000               3.60555               7.41620

 8.12404               6.92820               3.87298

 6.00000               8.00000               8.94427
```

6

Matrices and Vectors

Many array operations can only be performed using matrices or vectors. These are covered next.

MAT...IDN

The MAT... IDN statement establishes the specified matrix as an identity matrix: all elements in the matrix equal 0 except those in the main diagonal (upper left to lower right), which all equal 1.

MAT matrix name = IDN [(redim subscripts)]

An identity matrix must be square (two dimensions; each dimension has the same number of elements); when the subscripts are included, this enables the matrix to be redimensioned before the identity matrix is established.

Example

```
10   REM    This example shows MAT...IDN
20   OPTION BASE 1
30   DIM Identity(5,5)
40   MAT Identity=IDN(4,4)    ! Redimension matrix;
                                identity must be square
50   MAT PRINT Identity;
60   END
```

```
 1  0  0  0

 0  1  0  0

 0  0  1  0

 0  0  0  1
```

Matrix Multiplication

The **matrix multiplication** statement multiplies two matrices together. This is different from the multiplication of corresponding elements which was discussed previously.

MAT result matrix = operand matrix$_1$ * operand matrix$_2$

The number of columns of the first operand matrix **must** equal the number of rows of the second operand matrix. The result matrix has the same number of rows as the first operand matrix and the same number of columns as the second operand matrix. The result matrix can't be named the same as either of the operands. Here is an example –

$$B_{(3x4)} * C_{(4x2)} = A_{(3x2)}$$

Either or both of the operand matrices can also be vectors. The result matrix must also be a vector in this case. Here is an example –

$$X_{(5,6)} * Y_{(6)} = Z_{(5)}$$

If you have not been introduced to matrix multiplication, you might assume that corresponding elements are multiplied together; however, this is not the case. Assume you are multiplying matrix B by matrix C and storing the result into matrix A (MAT A=B*C). To determine the value of any element of matrix A, call it $A_{(x,y)}$, corresponding elements of the xth row of B and the yth column of C are multiplied together. The sum of the resultant products is the value for $A_{(x,y)}$.

Mathematically speaking –

$$\text{MAT A} = \text{B} * \text{C}$$

$$A(I,K) = \sum_{J=1}^{N} B(I,J) * C(J,K)$$

where N = the number of columns in B and rows in C

Example

Here's an example that uses matrix multiplication to find total sales for four bus routes using old and new prices –

Matrix A – Ticket Revenue by Route

Route	Single Trip	Round Trip	Commuter
1	143	200	18
2	49	97	24
3	314	77	22
4	82	65	16

Matrix B – Ticket Prices

	Old Price	New Price
Single Trip	.25	.30
Round Trip	.45	.50
Commuter	18.00	17.00

Matrix A, a 4 by 3 matrix, is multiplied by Matrix B, a 3 by 2 matrix, resulting in Matrix C, a 4 by 2 matrix.

Matrix C – Total Sales by Route

Route	Old	New
1	449.75	448.90
2	487.90	471.20
3	509.15	506.70
4	337.75	329.10

Here is the program used to perform the multiplication –

```
10   OPTION BASE 1
20   DIM A(4,3),B(3,2),C(4,2)
30   DATA 143,200,18,49,97,24,314,77,22,82,65,16
40   DATA 0.25,0.3,0.45,0.5,18,17
50   MAT READ A,B
60   MAT C=A*B
70   FIXED 2      ! For monetary output
80   MAT PRINT C
90   END
```

Here are some things to remember when using the matrix multiply statement –

- The result matrix can't be named the same as either of the two operand matrices.
- The number of columns of the first operand matrix must equal the number of rows of the second operand matrix.
- Either or both of the operand matrices can be a vector. In this case, the result matrix must also be a vector.

MAT...INV

The inverse of a square matrix can be found by using the MAT... INV statement –

MAT result matrix = INV operand matrix

If the determinant of the operand matrix (see the DET function) is 0, the matrix doesn't have an inverse. No warning is given to indicate this condition and a meaningless inverse is calculated. The best way to check the inverse is to multiply the original matrix by the inverse using matrix multiplication. The result should be close to an identity matrix.

The inverse of a matrix is useful for solving systems of equations.

Example

$$3X + 4Y = 47$$
$$2X + 2Y = 28$$

These two equations can be represented as matrices –

```
MAT C = A * B
```

$$\overset{A}{\begin{bmatrix} 3 & 4 \\ 2 & 2 \end{bmatrix}} \overset{B}{\begin{bmatrix} X \\ Y \end{bmatrix}} = \overset{C}{\begin{bmatrix} 47 \\ 28 \end{bmatrix}}$$

The solution (the values of X and Y) is determined by multiplying both sides of the equation by the inverse of A. The following program was used to solve the system of equations –

```
10   REM  Finding solution to system of 2 equations
20   OPTION BASE 1
30   DIM A(2,2),B(2,1),C(2,1),D(2,2)
40   DATA 3,4,2,2,47,28
50   MAT READ A
60   MAT READ C
70   MAT D=INV(A)
80   PRINT "MATRIX D, THE INVERSE OF MATRIX A:",D(*);
90   MAT B=D*C
100  PRINT "MATRIX B, THE VALUES OF X AND Y:",B(*)
110  END
```

```
MATRIX D, THE INVERSE OF MATRIX A:
-1   2

 1  -1.5


MATRIX B, THE VALUES OF X AND Y:
 9

 5
```

MAT...TRN

The transpose of a matrix can be found by using the MAT...TRN statement –

MAT result matrix = TRN operand matrix

The transpose of a matrix has the same elements as the original, but columns become rows, and rows become columns.

Example

```
10   REM  This example shows MAT...TRN
20   OPTION BASE 1
30   DIM A(2,4),B(4,4)      ! Must have 2 dimensions for
                              transposing an array
40   DATA 2,4,6,8,1,2,3,4
50   MAT READ A             ! Reads 8 values
60   MAT B=TRN(A)           ! Redimension B
70   PRINT "ORIGINAL ARRAY:",A(*);"TRANSPOSED:",B(*);
80   END
```

```
ORIGINAL ARRAY:
 2  4  6  8

 1  2  3  4


TRANSPOSED:
 2  1

 4  2

 6  3

 8  4
```

The result matrix is redimensioned, if necessary. The result and operand matrices must be separate matrices.

MAT...CSUM

The sums of all the columns of a matrix can be found by using the MAT...CSUM statement –

MAT result vector = CSUM operand matrix

Each element in the result vector is the sum of the corresponding column of the operand matrix.

Example

```
 10   REM   This examples shows MAT...CSUM
 20   OPTION BASE 1
 30   DIM Data(2,3),Sums(3)
 40   DATA 2,5,7,9,8,1
 50   MAT READ Data
● 60   MAT Sums=CSUM(Data)    ! Result is vector
 70   PRINT "ORIGINAL ARRAY:",Data(*);
 80   PRINT "COLUMN SUMS:";Sums(*);
 90   END
```

```
ORIGINAL ARRAY:
 2  5  7

 9  8  1


COLUMN SUMS:
 11  13  8
```

The result is redimensioned, if necessary.

MAT...RSUM

The sums of all the rows of a matrix can be found by using the MAT...RSUM statement –

MAT result vector = RSUM operand matrix

Each element in the result vector is the sum of the corresponding row of the operand matrix.

Example

```
 10   REM   This examples shows MAT...RSUM
 20   OPTION BASE 1
 30   DIM Data(2,3),Sums(3)
 40   DATA 2,5,7,9,8,1
 50   MAT READ Data
● 60   MAT Sums=RSUM(Data)    ! Result is vector
 70   PRINT "ORIGINAL ARRAY:",Data(*);
 80   PRINT "ROW SUMS:";Sums(*);
 90   END
```

```
ORIGINAL ARRAY:
 2  5  7

 9  8  1


ROW SUMS:
 14  18
```

The result vector is redimensioned, if necessary.

Array Functions

There are five array functions which each return a number that provides information about an array. These are covered next. Examples showing the array functions follow the descriptions of all the functions.

SUM Function

The SUM function returns the sum of all the elements in an array.

SUM operand array

ROW Function

The ROW function returns the number of rows in the array according to its current size. The number of rows corresponds to the subscript which is second from the right.

ROW operand array

COL Function

The COL (column) function returns the number of columns in the array according to its current size. The number of columns corresponds to the rightmost subscript.

COL operand array

A vector as the operand array always has one column.

DOT Function

The DOT function returns the inner (dot) product of two vectors.

DOT (vector name, vector name)

The two vectors must have the same working size. The inner product is the sum of the products of corresponding elements.

Example

A=2	B=1
4	2
6	4

DOT (A,B) = (2*1) + (4*2) + (6*4) = 34

The DET Function

The DET (determinant) function returns the determinant of the specified square matrix or of the last matrix which was inverted using the MAT...INV statement. If DET(operand matrix) results in zero, then the matrix doesn't have an inverse. No error is given if an inverse doesn't exist.

DET

DET operand array

If a matrix is not specified, the determinant of the last inverted matrix is returned. This method uses less memory because the determinant is a by-product of the inversion operation.

Examples

Here are some examples of array functions –

```
10   REM   This program shows DET function
20   OPTION BASE 1
30   DIM A(2,2),B(2,2)
40   INPUT "4 VALUES FOR MATRIX A?",A(*)
50   MAT B=INV(A)          ! Invert matrix A
60   IF DET=0 THEN 110     ! Use determinant of matrix A
70   PRINT "ORIGINAL MATRIX:",A(*)
80   PRINT "INVERSE OF MATRIX:",B(*)
90   PRINT "DETERMINANT OF INVERSE: ";DET(B)
100  STOP
110  PRINT "DETERMINANT IS 0; THERE IS NO INVERSE"
120  END
```

```
ORIGINAL MATRIX:
 4                    3

 2                    1


INVERSE OF MATRIX:
-.5                   1.5

 1                   -2


DETERMINANT OF INVERSE: -.5
```

```
 10   REM   This program shows ROW, COL and SUM functions
 20   OPTION BASE 1
 30   DIM Totals(5,5)
 40   REM  The following data items are daily totals
 50 Number_done:  DATA 4,2,5,6,9,7,8,10,5,16,44,15,18
 60 More_data:    DATA 1,5,9,12,4,6,18,7,10,5,8,7,4
 70   MAT READ Totals   ! This array is number of widgits
                          made each day by each employee
 80   INPUT "NUMBER OF EMPLOYEES TO BE TESTED?",E
 90   INPUT "NUMBER OF DAYS TO TEST?",D
 100  REDIM Totals(E,D)
● 110  PRINT ROW(Totals);" EMPLOYEES"
● 120  PRINT COL(Totals);" DAYS TESTED"
● 130  PRINT "TOTAL WIDGETS MADE IN TEST PERIOD:";SUM(Totals)
 140  END
```

```
 3  EMPLOYEES
 2  DAYS TESTED
TOTAL WIDGETS MADE IN TEST PERIOD: 33
```

Chapter 7

String Operations

A string is a series of ASCII characters such as 'vAB12?&', 'Mr. Smith', or '12 Oak Drive'. A string can be stored in a string variable like a number is stored in a numeric variable. Strings can be used for friendly, conversational programs and for text processing applications.

page 94 • **LET** (assigns a value to one or more strings)
page 94 • **DIM** (dimensions and reserves memory for strings – simple and array)
page 94 • **COM** (reserves space in a common memory area for strings - simple and array)
page 98 • **READ, MAT READ, DATA, INPUT, MAT INPUT** (used for assigning values to strings, just as for numeric variables)
page 98 • **LINPUT** (assigns any characters to a string from the keyboard)
page 99 • **EDIT** (lets the current value of a string be viewed and edited)

String Functions

LEN (current length)
POS (position of one string in another)
VAL (numeric value of a string of digits)
VAL$ (put a numeric value into a string)
CHR$ (convert numeric value to an ASCII character)
TRIM$ (remove all leading and trailing blanks)
UPC$ (convert all lowercase to uppercase)
LWC$ (convert all uppercase to lowercase)
RPT$ (repeat a string)
REV$ (reverse the characters in a string)
NUM (return the number which corresponds to a character)

Terms

- String name – a variable name followed by a dollar sign – $.
- Dimensioning – specifying the maximum number of characters of a string, within brackets – [ ].
- Implicit dimensioning – if a string is used, but not dimensioned, it is implicitly dimensioned to a maximum length of 18 characters.
- String array – a collection of strings, with each string being one element.
- String expression – text in quotes, string name substring, string concatenation, string function, user-defined string function or any combination of these.
- Substring – a part of a string made of zero or more contiguous characters, specified by placing substring specifiers in brackets after the string name. The three substring specifiers are –

  [character position] (the character and all following)
  [beginning character position, ending character position] (between and including the two characters)
  [beginning character position; number of characters] (the specified length, starting with the character)
- String concatenation – joining one string to the end of another using &.
- Null string – a string that contains no characters
- Literal – text within quotes

Overview

Like a numeric variable, each string must have a name which is followed by a dollar sign ($) to differentiate it from a numeric variable. Some examples of string names are –

```
Name$
Address$
City$
Response$
Title$
```

The simplest way to assign characters to a string variable is with the LET statement –

[LET] string variable [= string variable...] = "characters"

Any ASCII characters except quote marks can be assigned to a string this way. The quote marks which enclose the characters are not part of the string.

Example

```
10    LET Name$="Smith"     !  Quotes not part of string
20    Exclamation$="%$#?!*/"
30    STOP
```

Dimensioning a String

A string variable needs to be dimensioned before you use it. This can be done either implicitly or explicitly.

Explicit Dimensioning

The DIM and COM statements are used to dimension a string variable, specifying the **maximum** number of characters for the string. The value of a string can have any number of characters up to its maximum length. 32 767 is the longest any string can be. When a DIM or COM is executed, all strings are initialized to the null string, meaning they contain no characters.

See "Variables" in Chapter 5 for the amount of memory used.

When you dimension a string, you specify its maximum length in **brackets**[1]. Here are some examples –

```
10    OPTION BASE 1          ! Doesn't affect string length
20    DIM State$[20]         ! 20 characters maximum
30    DIM Address$[50]
40    COM Name$[40]
50    STOP
```

Implicit Dimensioning

If you use a string variable without having dimensioned it in a DIM or COM statement, it is implicitly dimensioned. The length of an implicitly dimensioned string is 18 characters.

String Arrays

A numeric array is a collection of numbers; similarly a string array is a collection of strings. Each string is one element in the array like a number is one element in a numeric array. It can be dimensioned in a DIM or COM statement. Every string in the array has the same maximum length. Like a numeric array, a string array can be implicitly dimensioned with 10 as the upper bound of each dimension.

In all string operations, an element of a string array can be used just like a simple string.

Example

```
10    OPTION BASE 1
20    DIM A$(2,3)[25]        ! String array: 6 elements of 25
                               characters each
30                           ! PARENTHESES for array dimensions
                               BRACKETS for string length
40    Data$(1,1,1)="SKI"     ! Implicit dimension of string array;
                               assign 'SKI' to 1 element (string)
50    A$(1,2)=A$(2,3)="**"   ! Assign '**' to 2 elements of array A$
60    PRINT Data$(1,1,1)     ! Output 1 string (1 element of Data$)
70    END
```

7

1 Brackets can also be obtained by shifting (and) in the numeric keypad.

String Expressions

Like you can with numbers, you can manipulate strings, creating a **string expression**. Text within quotes (a literal) is the simplest form of a string expression. The other forms that a string expression can have are discussed throughout this chapter. The kinds of string expressions are –

- Text within quotes
- String variable name
- Substring
- String concatenation operation
- String function
- User-defined string function

As with a numeric expression, a string expression can be enclosed in parentheses, if necessary.

Substrings

A substring lets you use any part of a string which is made up of zero or more contiguous characters. A substring is specified by placing **substring specifiers** in **brackets** after the string name. There are three forms a substring can have –

- String variable name [character position]

  The character position is a numeric expression which is rounded to an integer. The substring is made up of that character and all following it.

- String variable name [beginning character position, ending character position]

  This type of substring includes the beginning and ending characters and all in between. The character positions must be within the dimensioned number of characters. The ending character position must be greater than or equal to (beginning character position – 1). For example, A$[10,9] results in no output, but A$[10,8] results in an error.

- String variable name [beginning character position; number of characters]

  This type of substring begins with the specified character in the string and is the specified length. The number of characters specified can't exceed the dimensioned length, minus the beginning character position.

Example

```
10    OPTION BASE 1
20    DIM A$[25]
30    A$=" CARPENTRY"        ! Assign value to a string
                               note 1st character is a blank
40    PRINT "ORIGINAL STRING: ";A$,LIN(1)
50    PRINT "HERE ARE SOME SUBSTRINGS:",LIN(1)
60    PRINT "  A$[8]:";TAB(20);A$[8]
70    PRINT "A$[2,4]:";TAB(20);A$[2,4]
80    PRINT "A$[5;3]:";TAB(20);A$[5;3]
90    END
```

```
ORIGINAL STRING:  CARPENTRY

HERE ARE SOME SUBSTRINGS:

  A$[8]:              TRY
A$[2,4]:              CAR
A$[5;3]:              PEN
```

String Concatenation

The string concatenation operator joins (concatenates) one string to the back end of another.

string expression & string expression [& string expression...]

Example

```
10    A$="TRY"              ! Dimension A$ and B$ implicitly
20    B$="CAR"
30    PRINT B$&"PEN"&A$     ! Concatenate many strings
40    END
```

```
CARPENTRY
```

Assigning a Value to a String

LET, READ, MAT READ, ENTER, INPUT and MAT INPUT are used to assign value to string variables. If the string to be assigned is longer than the string variable's dimensioned length, error 18 (string too long) occurs. With the INPUT and LINPUT statements, error 18 causes the input routine to be repeated (the error cannot be trapped using ON ERROR).

Example

```
 10    !        Examples of READ, MAT READ, DATA
               using strings
 20    DATA Jones,Mikkelson,Miller
 30    DATA "A.","B.J.","M."
 40    OPTION BASE 1
 50    DIM Last_name$(3)[10],First_init$(3)[5]
 60    FOR I=1 TO 3
 70       READ Last_name$(I)
 80       PRINT Last_name$(I),
 90    NEXT I
 100   MAT READ First_init$
 110   PRINT First_init$(*),
 120   !          Names could be entered from the keyboard with
                  INPUT Last_name$(I)       for line 70
 130   !          and
                  MAT INPUT First_init$     for line 100
 140   END
```

```
Jones                 Mikkelson             Miller
A.
B.J.
M.
```

The LINPUT Statement

The LINPUT statement is used to assign any combination of characters to a string variable or substring –

LINPUT ["prompt" ,] string variable or substring

When LINPUT is executed, either a ? or the optional prompt is displayed. Characters typed in become the value of the string when STEP or CONT is pressed.

Example

```
 10    REM  This program shows LINPUT
 20    DIM Sentence$[50]
 30    Sentence$="When asked..."
 40    LINPUT "REPLY?",Sentence$[14]  ! Substring of Sentence$
 50    PRINT Sentence$
 60    END
```

If the response is –

```
"Compute, then print", he said
```

The output is –

```
When asked..."Compute, then print", he said
```

Pressing STEP or CONT without entering a value erases the current value of the string and sets it to the null string.

Note that the LINPUT statement allows an exclamation point or quotation mark to be included in the value of a string variable; this isn't possible with the INPUT statement.

The LINPUT statement can't be executed from the keyboard.

The EDIT Statement

The current value of a string can be viewed and edited by using the EDIT statement –

EDIT ["prompt",] string variable or substring

When the EDIT statement is executed, a ?, or the prompt if present, is displayed and the current value of the specified string appears in the keyboard entry area.

This value can then be edited like any keyboard entry. CLEAR LINE can be used to clear the line, allowing a totally new value to be entered, like with LINPUT. However, the original value can't be recalled. Pressing CONT stores the characters displayed in the keyboard entry area for the value of the string.

Example

EDIT could be used to alter the names in printed output –

```
 10   DIM O$[60]
 20   O$="Ed Smith spent "
 30   INPUT "AMOUNT SPENT",Dollars
 40   PRINT O$;Dollars
● 50   EDIT "NEW NAME",O$
 60   GOTO 30
 70   STOP
```

When line 50 is executed, NEW NAME is displayed. Ed Smith spent appears in the keyboard entry area. Then the character editing keys can be used to change the name.

The limit on the length of the string being edited is 160 characters (the length of the keyboard entry area). So, if a longer string is specified, ERROR 37 occurs. This can be avoided by using substrings.

Example

```
  10    DIM A$[200]
  20    A$=RPT$("1234567890",20)
• 30    EDIT "First 160 characters",A$[1,160]
• 40    EDIT "Last 40 characters",A$[161]
  50    PRINT A$
  60    STOP
```

The EDIT statement can't be executed from the keyboard.

String Variable Modification

A string or substring can be modified with a string expression. For example, a part of a string can be changed or characters can be added or deleted. The string containing the modification is called the **modifying string**; the string being modified is the **destination string**. The destination string can be a string or substring. The modifying string can be any string expression.

Example

```
  10    DIM A$[8]
  20    A$="GEOMETRY"
• 30    B$=A$&" BOOK"        ! B$ is destination string
                               A$&" BOOK" is modifying string
  40    PRINT B$
  50    END
```

```
GEOMETRY BOOK
```

If a modifying string is to be stored into a string or substring which is too short to hold it, the result is truncated on the right.

Each string of a string array can be modified in the same way as a simple string by the inclusion of subscripts.

No Substring Specifiers

When the destination string has no substring specifiers, the entire destination string is replaced by the modifying string or substring. Its characteristics after modification are the same as those of the modifying string or substring.

Example

```
  10    A$="HELLO"
  20    B$="GOODBYE"
  30    PRINT "A$= ";A$,"B$= ";B$
• 40    B$=A$              ! REPLACE B$
  50    PRINT "B$= ";B$
  60    END
```

```
A$= HELLO              B$= GOODBYE
B$= HELLO
```

One Substring Specifier

When the destination string has one substring specifier, the indicated substring is replaced by the modifying string or substring. The destination string can be shortened or lengthened.

Example

```
  1     DIM A$[25]
  10    A$="Atkins"
  20    PRINT A$
• 30    A$[3]="wood"
  40    PRINT A$
• 50    A$[3]="kinson"     ! string lengthened;  can't
                             go past maximum length
  60    PRINT A$
• 70    A$[3]="water"      ! string shortened
  80    PRINT A$
  90    END
```

```
Atkins
Atwood
Atkinson
Atwater
```

Characters added to those of a string must be contiguous; that is, they must immediately follow the destination string without any unassigned character spaces. If they are non-contiguous, ERROR 18 occurs.

Example

```
10    D$="Andy"
20    D$[8]="Atkinson"
30    PRINT D$
40    END
```

ERROR 18 in line 20 is caused because character positions 5, 6 and 7 aren't assigned any characters.

Two Substring Specifiers

When the destination string has two substring specifiers, with either a comma or semicolon, the indicated substring in the destination string is replaced by the modifying string expression. The left-most character of the modifying string expression replaces the left-most character of the indicated destination substring. The next adjacent character is replaced, and so forth, until the indicated destination substring is filled. If the modifying string is shorter than the indicated destination substring, the remainder of the destination substring is filled with blanks. If the modifying string is longer than the indicated destination string, the remainder of the modifying string is truncated.

Example

```
  10    A$="Loveland"
  20    PRINT A$
• 30    A$[1,4]="Home"
  40    PRINT A$
• 50    A$[1,4]="Up"            ! Modifying string shorter
                                  pad with blanks
  60    PRINT A$
• 70    A$[1,4]="Tomorrow"      ! Modifying string longer
                                  truncate on the right
  80    PRINT A$
  90    END
```

```
Loveland
Homeland
Up  land
Tomoland
```

The length of the destination string after modification either is unchanged, or is greater. When the value of the second substring specifier is greater than the current length of the destination string, the modification results in a lengthened string (within its maximum length).

Example

```
10   C$="Goodbye"
20   PRINT C$
30   C$[5,9]="times"     ! C$ is lengthened
40   PRINT C$
50   END
```

```
Goodbye
Goodtimes
```

The Null String

The null string is a string which contains no characters or blanks. It can be used to erase the value of a string, or to check to see if a string has a value. The following examples each specify the null string –

```
10   LET N$ = ""
20   M$ = A$[4,3]
```

All strings are initialized to the null string by a DIM or initial COM statement or when SCRATCH V or SCRATCH C is executed. The null string can be used to clear a string.

String Functions

String functions let you manipulate a string. They are especially useful in text processing applications.

The LEN Function

The length (LEN) function returns the number of characters in a string expression –

LEN string expression

The current length of the string expression is returned.

Example

```
10   !      This example uses LEN function to center
            a line of output
30   PRINTER IS 0,WIDTH(40)
40   Area$="KEYSTONE"
50   C=(40-LEN(Area$))/2    ! Finds centering factor
                              using printer width of 40
60   PRINT "*";TAB(40),"*"
70   PRINT TAB(C),Area$     ! Centers output
80   PRINT "*";TAB(40),"*"
90   END
```

```
*                                        *
                KEYSTONE
*                                        *
```

The POS Function

The position (POS) function determines the position of a substring within a string –

POS (in string expression, of string expression)

If the second string expression is contained within the first, the value returned is the position of the first character of the second string expression within the first string expression. If the second string expression is **not** contained within the first string expression, or if the second string expression is the null string, the value returned by the function is 0. If the second string expression occurs in more than one place within the first string expression, only the first occurrence is used by the function.

Example

```
10   !   This example shows the POS function
20   PRINT "POSITION OF 'PARK' IN 'WINTER PARK':";
30   PRINT POS("WINTER PARK","PARK")
40   PRINT "POSITION OF 'PARKS' IN 'WINTER PARK':";
50   PRINT POS("WINTER PARK","PARKS")
60   !   These lines uses POS to extract name from sentence
70   DIM A$[50]
80   A$="MR. SMITH SPENT EIGHTY DOLLARS"
90   Start=POS(A$,".")        ! Name is after period
100  End=POS(A$,"SPENT")      ! Name is before 'SPENT'
110  PRINT A$[Start+1,End-1]
120  END
```

```
POSITION OF 'PARK' IN 'WINTER PARK': 8
POSITION OF 'PARKS' IN 'WINTER PARK': 0
 SMITH
```

The VAL Function

With the value (VAL) function, a string or a substring containing digits, including any exponent, can be used in calculations. (Normally the characters in a string are not recognized as numeric data and can't be used in numeric calculations.)

VAL (string expression)

The first character to be converted in a string using the VAL function must be a digit, a plus or minus sign, a decimal point or a space. A leading plus sign or space is ignored; a leading minus sign is taken into account. All following characters must be digits, a decimal point or an E. An E character after a numeric and followed by digits or a plus or minus sign and digits is interpreted as exponent of base 10. A decimal point following digits after an E terminates the exponent.

Numeric data entries can be combined logically with input text. All contiguous numerics are considered a part of the number until a non-numeric is reached in the string. This means that a string can contain more than one number. The first character of the string expression after leading spaces, plus signs or minus signs must be a digit or a decimal point. If the leading part of the string is not a valid number, ERROR 32 occurs.

Example

```
 10   REM    Examples of VAL function
 30   A$="JONES,J: 291228811"
 40   PRINT VAL(A$[10])     ! Using social security
                              number from string
 50   PRINT VAL("1E2.5")
 60   END
```

```
291228811
100
```


The VAL$ Function

The VAL$ function is (nearly) the inverse of the VAL function and returns a string representing the number, in the current number format (STANDARD, FIXED or FLOAT) without leading or trailing blanks –

VAL$ (numeric expression)

Example

```
10   !    This example shows VAL$ function
20   Name$="B. JONES"
30   Age=28
40   Info$=Name$&" IS "&VAL$(Age)
50   PRINT Info$
60   !    The next 2 lines show that the value that is
          used by VAL$ depends on the current output mode
70   FLOAT 3
80   PRINT LIN(2),"VAL$(120)= ";VAL$(120)
90   END
```

```
B. JONES IS 28


VAL$(120)= 1.200E+02
```

The CHR$ Function

The character (CHR$) function converts a numeric value in the range −32 768 through 32 767 into a string character. Any number out of the range 0 through 255 is converted MOD 256 to that range. Any 8-bit character code can be stored in a string using the character function which is especially useful for accessing control codes and putting quotes into a string.

CHR$ (numeric expression)

Example

```
10   !   Using CHR$ function to put quotes in a string
20   DIM A$[50]
30   A$="BILL SAID, "&CHR$(34)&"LET'S GO!"&CHR$(34)
40   PRINT A$
50   END
```

```
BILL SAID, "LET'S GO!"
```

See the Reference Tables for the ASCII table of correspondence between characters and numbers in the range 0 through 127. Using this function with numbers in the range 128 through 159 is useful for output of CRT special features (inverse video, blinking, underline); see Appendix A. Numbers in the range 160 through 255 are used to access national and drawing characters; see the Reference Tables.

The NUM Function

The numeric (NUM) function converts an individual string character to its corresponding value, represented decimally.

NUM (string expression)

The decimal equivalent of the first character of the expression is returned.

Example

```
10    !     Using NUM function to find number in a string
            VAL function is used to output the number
20    DIM A$[50]
30    A$="THE BOSS'S NUMBER IS 5555555"
40    FOR I=1 TO LEN(A$)
50       IF (NUM(A$[I])>=48) AND (NUM(A$[I])<=57) THEN 80
60       ! Branch when a digit is encountered
70    NEXT I
80    PRINT "PHONE NUMBER STARTS IN POSITION";I;" OF A$"
90    PRINT "PHONE NUMBER IS:";VAL(A$[I])
100   END
```

```
PHONE NUMBER STARTS IN POSITION 22  OF A$
PHONE NUMBER IS: 5555555
```

The UPC$ Function

The uppercase (UPC$) function returns a string with all lowercase ASCII letters converted to uppercase.

UPC$ (string expression)

Example

```
10    !       Using UPC$ function with INPUT response
20    DIM Answer$[25]
30    INPUT "DO YOU WISH TO CONTINUE?",Answer$
40    IF UPC$(Answer$)="NO" THEN STOP
50    IF UPC$(Answer$)="YES" THEN 90
60    BEEP
70    PRINT "I DON'T UNDERSTAND YOUR ANSWER; ANSWER AGAIN"
80    GOTO 30
90    REM  Rest of program follows ...
100   STOP
```

The uppercase function allows strings to be compared without regard to upper and lowercase. This is useful for standardizing input responses.

The LWC$ Function

The lowercase (LWC$) function returns a string with all uppercase ASCII letters converted to lowercase –

LWC$ (string expression)

Example

```
10   !    Using LWC$ function for alphabetizing
20   INPUT "NUMBER OF NAMES?",N
30   FOR I=1 TO N
40       INPUT "NAME?",Name$(I)
50       Name$(I)=LWC$(Name$(I))   ! Puts name into lowercase
60   NEXT I
70   !    Alphabetizing sequence would follow ...
```

THE RPT$ Function

The repeat (RPT$) function allows a string of characters to be repeatedly concatenated –

RPT$ (string expression, number of repetitions)

Example

```
10   REM  Using RPT$ function to box a title
20   INPUT "TITLE OF BOOK?",Title$
30   PRINT " "&RPT$("_",LEN(Title$)),LIN(1)
40   PRINT "|"&Title$&"|"
50   PRINT " "&RPT$("_",LEN(Title$)),LIN(1)
60   END
```

```
 ____________________
|GONE WITH THE WIND|
 ____________________
```

The number of repetitions can be any numeric expression in the range 0 through 32 767 when rounded. If 0 is specified, the result is the null string. The length of the result can't exceed 32 767 characters.

The REV$ Function

The reverse (REV$) function reverses the order of the characters in a string –

REV$ (string expression)

Example

```
10    !     Using REV$ to output a line of text
            without splitting a word in the middle
20    PRINTER IS 16,WIDTH(40)
30    DIM Line$[80],Temp$[80]
40    Line$[1,28]="STRING FUNCTIONS ARE USEFUL"
50    Line$[29]="FOR TEXT PROCESSING APPLICATIONS"
60    Temp$=REV$(Line$[1,40])        ! Reverse 1st 40 characters
70    FOR I=1 TO 40
80       IF Temp$[I,I]=" " THEN 100! Search for blank
90    NEXT I
100   PRINT Line$[1,40-I+1]          ! Output up to blank
110   PRINT Line$[40-I+2]            ! Output from blank on
120   END
```

```
STRING FUNCTIONS ARE USEFUL FOR TEXT
PROCESSING APPLICATIONS
```

The TRIM$ Function

The trim (TRIM$) function deletes leading and trailing blanks from a string –

TRIM$ (string expression)

Example

```
10    !     Examples of TRIM$ function
20    PRINT "*"&TRIM$("   STRING   "&"   FUNCTIONS    ")&"*"
30    !     Note that internal blanks aren't trimmed
            TRIM$ is useful for trimming INPUT responses
40    END
```

```
*STRING      FUNCTIONS*
```

7

Relational Operations – Comparing Strings

String variables may be compared using the relational operators –

=	<
>	<=
>=	<> or #

Each character in a string is represented by a standard equivalent decimal code, as shown in the ASCII Table in the Reference Tables. When two string characters are compared, the lesser of the two characters is the one whose decimal code is smaller. For example, 2 (decimal code 50) is smaller than R (decimal code 82).

Strings are compared, character by character, from left to right until a difference is found. If one string ends before a difference is found, the shorter string is considered the lesser. For example, "STEVE" is smaller than both "STEVEΔ" and "STEVEN".

Examples

Here is an example which could be used to allow communication between the computer and the user –

```
  10   DIM Answer$[20]
  20   INPUT "DO YOU WISH TO CONTINUE--YES OR NO?",Answer$
● 30   IF Answer$="NO" THEN STOP
● 40   IF Answer$<>"YES" THEN 20
  50   REM   Rest of program follows ...
```

In some cases, such as in alphabetic sequencing problems, it is useful to compare strings for conditions other than "equal to" and "not equal to". For example, to arrange several different strings in alphabetical order, the following type of string comparison could be included in a program.

```
  10   DIM Name1$[20],Name2$[20],Temp$[20]
  20   INPUT "TWO NAMES?",Name1$,Name2$
● 30   IF Name1$<Name2$ THEN 70
  40   Temp$=Name2$          ! TEMPORARY STRING USED
  50   Name2$=Name1$
  60   Name1$=Temp$
  70   PRINT Name1$;" COMES BEFORE ";Name2$
  80   END
```

```
JOHNSON COMES BEFORE JONES
```

Computer Museum

Variable Diagram

The variable diagram below shows how string variables are related to each other and to numeric variables.

- variable
  - single variable
    - numeric variable
      - simple numeric
      - array element
    - string variable
      - simple string
        - substring
      - string array element
        - substring
  - array variable
    - numeric array
      - matrix
      - vector
    - string array
      - matrix
      - vector

Memory Usage

In memory, a simple string uses 6 bytes + 1 byte per character in the current length (rounded up to an even integer).

A string array uses 12 bytes + 4 bytes per dimension + 2 bytes per element + 1 byte per character (rounded up to an even integer) in each string of the array.

Chapter 8
Branching and Subroutines

page 114 • **GOTO** (transfers execution to the specified line)
page 114 • **ON...GOTO** (transfers execution to one of one or more lines, depending on the value of the numeric expression)
page 115 • **IF...THEN** (causes branching or execution of a statement if a condition is true)
page 117 • **FOR,NEXT** (form a loop of the statements between them, which is executed a specified number of times)
page 122 • **GOSUB** (transfers execution to the subroutine that starts at the specified line)
page 123 • **ON...GOSUB** (transfers execution to one of one or more subroutines, depending on the value of the numeric expression)
page 122 • **RETURN** (last line of a subroutine; transfers execution to the line after the GOSUB)
page 125 • **DEF FN** (defines a numeric or string user-defined function)
page 125 • **FN** (accesses a user-defined function)

Terms

- Loop – the statements enclosed by FOR and NEXT that are executed repeatedly

```
FOR
.
.
.
NEXT
```

- Nesting – placing one loop completely within another

```
FOR I
    FOR J
    .
    .
    .
    NEXT J
NEXT I
```

- Subroutine – a group of statements that performs a task, is accessed with GOSUB and ends with RETURN
- User-defined function – a function you define with an expression and give a name. It returns a value.

Unconditional Branching

The GOTO and ON...GOTO statements provide unconditional branching by transferring control to a specified line.

The GOTO Statement

The GOTO statement specifies a higher or lower-numbered line in the same program segment, where execution is to be transferred –

GOTO line identifier

Example

```
  10    PRINT "THIS IS LINE 10"
• 20    GOTO 40                  ! Could also say GOTO Line_40
  30    PRINT "THIS IS LINE 30"
  40 Line_40:  PRINT "THIS IS LINE 40"
  50    END
```

```
THIS IS LINE 10
THIS IS LINE 40
```

The ON...GOTO Statement

The ON...GOTO (computed GOTO) statement allows control to be transferred to one of one or more statements in the same program segment based on the value of a numeric expression –

ON numeric expression GOTO line identifier list

The numeric expression is evaluated and rounded to an integer. A value of 1 causes control to be transferred to the first line identifier in the list; a value of 2 causes control to be transferred to the second line identifier in the list, and so on.

Example

```
  10    INPUT "FULL, LOW OR OUT-OF-STOCK?(1,2 OR 3)",Report
• 20    ON Report GOTO 30,50,Problem
  30    PRINT "FULL STOCK"
  40    STOP
  50    PRINT "STOCK IS LOW; IT NEEDS TO BE WATCHED"
  60    STOP
  70 Problem:    PRINT "NO STOCK LEFT--RE-ORDER!!!"
  80    END
```

If the value of the numeric expression is less than 1 or greater than the number of line identifiers in the list, ERROR 19 (improper value) occurs.

Summary

Here are some facts to remember concerning the GOTO statements –

- All lines specified by GOTO statements must be in the same program segment. Otherwise, ERROR 3 occurs.
- If the line specified as the destination of a branch is not an executable statement[1], program control is transferred to the first executable statement following the specified line. However, execution pauses at the specified line if STEP is being used.
- GOTO statements are programmable only; they can't be executed from the keyboard.

The IF...THEN Statement

The IF...THEN statement is used to provide branching which is dependent on a specified condition –

IF expression THEN line identifier

Branching occurs if the expression evaluates logically as true. If the numeric expression has a value other than 0, it is considered true and branching to the specified line occurs. If it has a value of 0 (false), execution continues with the line following the IF...THEN statement.

Examples

```
  10   INPUT A
• 20   IF A THEN 50          ! Branching only if A is not 0
  30   PRINT "A=0"
  40   GOTO 10
  50   PRINT "A IS NOT 0 -- A=";A
  60   END
```

8

1 The following statements are declaratory, non-executable statements: COM, DATA, DEF FN, DIM, END, FN END, IMAGE, INTEGER, OPTION BASE, REAL, REM, SHORT, SUB, SUBEND.

The IF...THEN statement is used most often with relational and logical operators.

```
 10   Pay=4.75
 20   INPUT "HOURS WORKED?",Hours
● 30   IF Hours>40 THEN Overtime
 40   DISP "NO OVERTIME"
 50   WAIT 2000
 60   GOTO 20
 70 Overtime:    Overtime=Hours-40
 80   PRINT "OVERTIME PAY IS:";Overtime*Pay*1.5
 90   STOP
```

```
 10   INPUT "PRINTER SELECT CODE?",Psc
● 20   IF (Psc<0) OR (Psc>16) OR (Psc=13) THEN Error
● 30   IF (Psc=14) OR (Psc=15) THEN Error
 40   !    Lines 20 and 30 check for invalid select code
 50   PRINTER IS Psc
 60   PRINT "PRINTER IS SELECT CODE";Psc
 70   STOP
 80 Error: PRINT Psc;"IS INVALID SELECT CODE; ENTER ANOTHER ONE"
 90   BEEP
 100  GOTO 10
```

Another form of the IF...THEN statement provides conditional execution of a statement without necessarily branching –

IF numeric expression THEN statement

When the value of the numeric expression is not equal to 0 (true) the statement is executed. When the value of the numeric expression is 0 (false), execution continues with the following line.

Example

```
 10   INPUT "ENTER VALUES FOR X AND Y",X,Y
● 20   IF X=Y THEN PRINT "X=Y=";X
● 30   IF X<>Y THEN PRINT "X IS NOT EQUAL TO Y"
 40   END
```

All BASIC statements are allowed after THEN with the following exceptions –

FOR	IMAGE	NEXT	INTEGER
IF	OPTION BASE	DATA	REAL
COM	REM	DIM	SHORT
DEF FN	SUB	FN END	SUBEND
END			

The FOR and NEXT Statements

Repeatedly executing a series of statements is known as **looping**. The FOR and NEXT statements are used to enclose a series of statements in a **FOR-NEXT loop**, allowing them to be repeated a specified number of times.

FOR loop counter = initial value TO final value [STEP increment value]

.

.

.

NEXT loop counter

The FOR statement defines the beginning of the loop and specifies the number of times the loop is to be executed. The loop counter must be a simple numeric variable.

The initial, final, and increment values can each be any numeric expression. If the increment value is not specified, the default value is 1.

Examples

Here's an example of a FOR-NEXT loop –

FOR-NEXT loop range

```
10   FOR I=1 TO 5          ! First statement of loop
20      PRINT "I EQUALS";
30      PRINT I
40      REM   Indenting the body of the loop makes
              the program easier to read
50   NEXT I                ! Last statement of loop
60   PRINT "FINISHED WITH THE LOOP;  I NOW EQUALS";I
70   END
```

```
I EQUALS 1
I EQUALS 2
I EQUALS 3
I EQUALS 4
I EQUALS 5
FINISHED WITH THE LOOP;  I NOW EQUALS 6
```

In this example, I is established as the loop counter and is set to 1 when the FOR statement is executed. The FOR-NEXT loop is executed 5 times – when I = 1, 2, 3, 4 and 5. Each time the NEXT statement is executed, the value of I is incremented by 1, the default increment value. When the value of I exceeds the final value (when I = 6) the loop is finished and execution continues with the statement following the NEXT statement.

The following examples show that differing FOR statements can perform the same task. In each example, the FOR-NEXT loop is executed ten times. Notice the value of the loop counter while the loop is executing and after it is complete.

```
● 10   FOR A=2 TO 12
  20   NEXT A
  30   PRINT "A=";A
  40   END
```

```
  A= 13
```

```
  10   X=10
  20   Y=100
● 30   FOR A=X TO Y STEP 10   ! Initial & final values
                                can be expressions
  40   NEXT A
  50   PRINT "A=";A
  60   END
```

```
  A= 110
```

```
● 10   FOR A=10 TO 1 STEP -1    ! It is easy to
                                  decrement the counter
  20   NEXT A
  30   PRINT "A=";A
  40   END
```

```
  A= 0
```

```
 10   X=2
● 20   FOR A=1 TO 19 STEP X    ! Value of counter is 1,3,5, etc.
                                 STEP value can be expression
 30   NEXT A
 40   PRINT "A=";A
 50   END
```

```
A= 21
```

Programming Hint

An often overlooked aspect of FOR-NEXT looping is that the **actual** value of the counter when the loop is complete does not equal the **final** value.

Computer Museum

The advantages of using FOR-NEXT looping instead of an IF...THEN statement are shown in the following examples where the numbers 1 through 1000 are printed in succession. The program that uses the FOR-NEXT loop is easier to key in and uses less memory.

IF...THEN statement

```
 10   I=1
 20 Loop:  IF I>1000 THEN End   ! Should loop be executed again?
 30   PRINT I
 40   I=I+1                     ! Increments the counter
 50   IF I<=1000 THEN Loop        ! Back to beginning of loop
 60 End:   END
```

FOR-NEXT loop

```
 10   FOR I=1 TO 1000
 20   PRINT I
 30   NEXT I      ! Increment counter, test to see if loop should
                    be executed again, branch to start of loop
 40   END
```

8

The initial, final and increment values are calculated upon entry into the loop; the calculated values are used throughout execution of the loop. The following example illustrates that the initial, final and increment values can be changed without affecting the number of times the loop is repeated.

```
 10   A=3
 20   INPUT B
● 30   FOR X=A TO A*B STEP B-2  ! STEP value can be an expression
 40      A=A+X
 50      B=B-1                 ! Lines 40 & 50 don't affect initial
                                 and final values of the counter
 60      PRINT X,A,B
 70   NEXT X
 80   END
```

If 4 is input for the value of B, the loop is repeated 5 times and the output is –

```
 3                    6                    3
 5                    11                   2
 7                    18                   1
 9                    27                   0
 11                   38                  -1
```

Nesting

FOR-NEXT loops can be nested. When one loop is contained within another, the inner loop is said to be nested. The following example illustrates assigning values to an array using a nested FOR-NEXT loop.

```
 10   OPTION BASE 1
 20   DIM Array(4,3)
● 30   FOR X=1 TO 4      ! Outer loop controls 1st (left) subscript
 40     FOR Y=1 TO 3    ! Inner loop controls 2nd (right) subscript
 50       Array(X,Y)=X+Y
 60     NEXT Y
 70   NEXT X
 80   MAT PRINT Array
 90   END
```

```
 2                    3                    4

 3                    4                    5

 4                    5                    6

 5                    6                    7
```

8

A FOR-NEXT loop can not overlap another.

Correct Nesting

```
10   FOR I=1 TO 3          ! Beginning of outer loop
20      FOR J=4 TO 6       ! Beginning of inner loop
30         PRINT I,J
40      NEXT J             ! End of inner loop
50   NEXT I                ! End of outer loop
60   END
```

Incorrect Nesting

```
10   FOR I=1 TO 3          ! Beginning of outer loop
20      FOR J=4 TO 6       ! Beginning of inner loop
30         PRINT I,J
40   NEXT I                ! End of outer loop
50      NEXT J             ! End of inner loop
60   END
```

In the incorrect nesting example, the I loop is activated and then the J loop is activated. The J loop is cancelled when `NEXT I` is executed because it's an inner loop. When the I loop is completed and `NEXT J` is accessed, `ERROR 6 IN LINE 50` is displayed. This is because the J loop was cancelled and was not reactivated after the last I loop.

FOR-NEXT Loop Considerations

Execution of FOR-NEXT loops should always start with the `FOR` statement. Branching into the middle of a loop produces `ERROR 6` when `NEXT` is executed, because no corresponding `FOR` statement was executed.

Execution of a loop normally ends with the `NEXT` statement. It is permissible to transfer control out of the loop using a statement within the loop. After an exit is made through this method, the current value of the counter is retained and is available for later use in the program. After leaving a FOR-NEXT loop, it is permissible to re-enter the loop either at a statement within the loop, or at the `FOR` statement, thereby reinitializing the counter.

The FOR and NEXT statements execute faster if the loop counter is an integer.

Subroutines

Many times, the same sequence of statements is executed in many places within a program. A subroutine allows the group of statements to be keyed in only once and to be accessed from different places in a program. A subroutine return pointer is kept by the system in the execution stack to indicate where execution is to return to when the subroutine is complete. The GOSUB and ON...GOSUB statements are used to access subroutines.

The GOSUB Statement

The GOSUB statement transfers control to the subroutine which begins at the specified line in the same program segment –

GOSUB line identifier

A subroutine ends logically with the RETURN statement –

RETURN

which transfers control back to the statement immediately following the GOSUB statement.

Example

Here is an example of accessing a subroutine from different places in a program –

Subroutine

```
 10   INPUT "HOURS WORKED?",Hours
 20   PRINT "TOTAL HOURS WORKED IS";Hours
 30   Rate=5.25
 40   Overtime=.25
●50   GOSUB Comp    ! Branch to subroutine at line 80
 60   PRINT "TOTAL PAY AT 5.25/HOUR IS";Pay
 70   GOTO 120      ! Branch around subroutine
 80 Comp: Pay=Hours*Rate   ! First line of subroutine
 90        Ohours=Hours-40
100        Pay=Pay+Ohours*Overtime
110        RETURN        ! Last line of subroutine; control
                           passes to line after GOSUB
120   Rate=6.00
130   Overtime=.29
●140  GOSUB Comp        ! Branch to subroutine at line 80
150   PRINT "TOTAL PAY AT 6.00/HOUR IS";Pay
160   END
```

```
TOTAL HOURS WORKED IS 43
TOTAL PAY AT 5.25/HOUR IS 226.5
TOTAL PAY AT 6.00/HOUR IS 258.87
```

The ON...GOSUB Statement

The ON...GOSUB (computed GOSUB) statement allows any of one or more subroutines in the same program segment to be accessed based on the value of a numeric expression –

ON numeric expression GOSUB line identifier list

The numeric expression is evaluated and rounded to an integer. A value of 1 causes the subroutine specified by the first identifier in the list to be accessed; a value of 2 causes the subroutine specified by the second identifier in the list to be accessed, and so on.

Example

```
 10   FOR X=1 TO 3
• 20   ON X GOSUB One,Two,Three ! Each pass through loop gets
                                  a different subroutine
 30   NEXT X
 40   STOP                 ! STOP prevents accessing of subroutines
 50 One:  PRINT "FIRST SUBROUTINE"
 60   RETURN
 70 Two:  PRINT "SECOND SUBROUTINE"
 80   RETURN
 90 Three:  PRINT "THIRD SUBROUTINE"
 100  RETURN
 110  END
```

```
FIRST SUBROUTINE
SECOND SUBROUTINE
THIRD SUBROUTINE
```

If the value of the numeric expression is less than 1 or greater than the number of line identifiers in the list, ERROR 19 occurs.

A second subroutine can be entered before the RETURN of the first is executed.

Example

```
  10    INPUT "INPUT VALUES FOR X AND Y",X,Y
• 20    GOSUB Try
  30    STOP
  40 Try: PRINT X,Y
  50       IF X<Y THEN GOSUB Nest ! Branch to 2nd subroutine
  60       PRINT "X*Y=";X*Y
  70       RETURN
• 80 Nest:    PRINT "X<Y"          ! First line of 2nd subroutine
  90          RETURN
  100   END
```

The subroutine at line 70 is accessed before the one at line 40 is completed.

Subroutines can be accessed in this manner as much as available memory allows. Doing it too many times can cause the execution stack to become too large, thus causing a memory overflow. See Appendix F for more information. When a RETURN is executed, control returns to the line following the most recently executed GOSUB.

Summary

Here are some facts to remember concerning subroutines and the GOSUB statements –

- A subroutine should always end with a RETURN statement.
- GOSUB statements are programmable only; they can't be executed from the keyboard.
- All subroutines specified must be in the same program segment.

THE DEF FN Statement

If a numeric or string operation has to be evaluated several times, it is convenient to define it as a function. This is done using the DEF FN statement which specifies a user-defined function, returns a single value as the value of the function and can be used like a system function.

The simplest form is the single-line function which can be used to define a numeric or string function (there is also a multiple-line function; see Chapter 9).

These two statements are used for defining a numeric (first syntax) and a string (second syntax) function –

- DEF FN function name [(formal parameter list)][1] = numeric expression
- DEF FN function name $ [(formal parameter list)] = string expression

The function name must follow the rules of a valid name. The expression can include both parameters[1] and variables.

Once the function is defined, you reference it and supply values by using the following syntax. The first syntax references a numeric function, the second a string function.

- FN function name [(pass parameter list)[1]]
- FN function name $ [(pass parameter list)]

When the function reference, FN, is encountered, control is transferred to the corresponding DEF FN. The values of the pass parameters are substituted for the formal parameters and the expression is evaluated. Its value is returned as the value for the referencing syntax. See Chapter 9 for a more detailed explanation of parameters.

NOTE

Single-line functions are local to the program segment in which they are defined. The DEF FN statement can't contain a reference to itself. Otherwise ERROR 48 occurs.

8

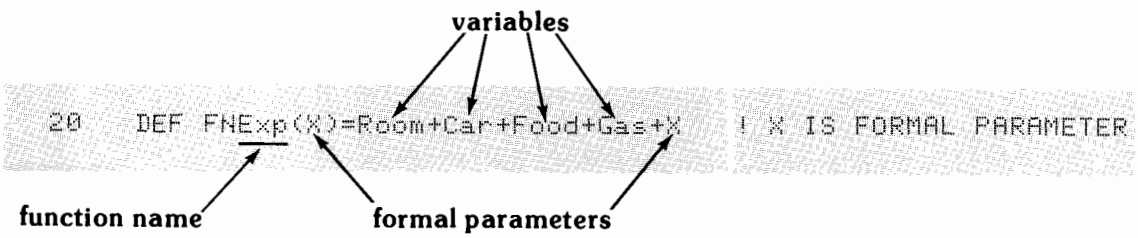
1 Parameters, formal and pass parameter lists are discussed in Chapter 9.

Example

Here's an example use of a single-line function. Say that a program contains these lines –

```
30   Expenses(I)=Room+Car+Food+Gas+Ticket
80   Expenses(I)=Room+Car+Food+Gas+Phone
200  Expenses(I)=Room+Car+Food+Gas+Ski_rental
```

By defining –



lines 30, 80, and 200 can be simplified –

```
30   Expenses(I)=FNExp(Ticket)        ! Ticket IS PASS PARAMETER
80   Expenses(I)=FNExp(Phone)
200  Expenses(I)=FNExp(Ski_rental)
```

referencing syntax

Summary

Here are some facts to remember when using single-line functions –

- The name of the function must be in the form of a valid name.
- The expression used to define the function can contain both variables and formal parameters.
- A single-line function can't be recursive; that is, it can't contain a reference to itself.
- Single-line functions are local to the program segment in which they are defined. That means that they can't be accessed from any other program segment.

Chapter 9
Subprograms

page 133 • **DEF FN** (the first line of a multiple-line user-defined function)
page 133 • **FN END** (the last line of a multiple-line user-defined function)
page 133 • **RETURN** (specifies the value to be returned for a function and transfers execution back)
page 133 • **FN** (accesses a user-defined function)
page 136 • **SUB** (the first line of a subroutine subprogram)
page 136 • **SUBEND** (the last line of a subroutine subprogram and transfers execution back)
page 136 • **SUBEXIT** (transfers execution back from a subroutine subprogram)
page 136 • **CALL** (accesses a subroutine subprogram)

Terms

- Subroutine subprogram – a separate program segment that performs a task under the control of the calling program segment.
- Multiple-line user-defined function subprogram – a separate program segment that returns a single numeric or string value to the calling program segment and is used like a system function such as SIN.
- Main program – the central part of a program which is accessed when you press RUN
- Program segment – The main program and each subprogram are all known as program segments.
- Formal parameters – used to define subprogram variables; can be simple variables, array identifiers and #file number.
- Pass parameters – used to pass values from the calling program segment to the subprogram; can be simple variables, array identifiers numeric expressions and #file number.
- Pass by reference – letting the formal and pass parameters share memory which lets the value of the calling program variable be changed within the subprogram.
- Pass by value – letting the subprogram variables have their own temporary memory so that calling program variables can't be changed. Enclosing a pass parameter in parentheses lets it be passed by value.
- Local variable – a variable in a subprogram that isn't in the formal parameter list or subprogram COM statement.

Why Use Subprograms?

Many programs include various routines that require a long series of statements (such as routines for sorting or computing compound interest) that must sometimes be repeated many times using different values in one program. To avoid rewriting a routine each time it is needed, a **subprogram** can be used.

A subprogram is a set of statements that performs a certain task under the control of the calling program segment. It differs from a subroutine in that it is a separate segment, coming after the main program.

A subprogram enables you to repeat an operation many times, substituting different values each time the subprogram is called. Subprograms can be called at almost any point in a program, and are convenient and easy to use. They can give greater structure and independence to a program. They can also be used to save memory through the use of local variables. A main program may be a sort of "skeleton" program which calls many subprograms, which, in turn, can call other subprograms.

Types of Subprograms

There are two types of subprograms.

- The **function subprogram** (or multiple-line user-defined function) is designed to return a single numeric or string value to the calling program and is used like system functions such as SIN or CHR$. It is defined using the DEF FN statement.
- A **subroutine subprogram** is designed to perform a specific task under the control of the calling program segment. It is defined using the SUB statement. A subroutine subprogram is similar to a subroutine subprogram in FORTRAN.

Terms

There are a few terms which are important to know when dealing with subprograms.

Main program – The central part of a program from which subprograms can be called is known as the main program. When you press (RUN), you access the main program. The main program can't be called by a subprogram.

Program segment – The main program and each subprogram are known as program segments. Every program segment is independent of every other program segment. Subprograms come after the main program; that is, they have higher line numbers. Subprograms are called by the main program or another subprogram. See "Memory" in the Reference Tables for the relationship between memory allocation and subprograms.

Calling program – When a subprogram is being executed, the program segment (main program or subprogram) which called the subprogram is known as the calling program. Control returns to the calling program when the subprogram is completed.

Current environment – The program segment which is being executed is known as the current environment.

Parameters

Values are passed between a subprogram and the calling program using parameters. There are two kinds of parameters. **Formal parameters** are used in defining the subprogram. **Pass parameters** are used to pass values from the calling program to the subprogram. Each pass parameter corresponds to a formal parameter.

Formal Parameters

The formal parameter list is used in a SUB or DEF FN statement to define the subprogram variables, and to relate them to calling program variables. It can include non-subscripted numeric and string variable names, array identifiers and file numbers in the form: # file number. Parameters must be separated by commas and the parameter list must be enclosed in parentheses.

Numeric type – REAL, SHORT, INTEGER – can be declared in a formal parameter list by placing the type word before a parameter or group of parameters.

Example

Here are examples of some formal parameter lists –

```
270  DEF FNPay(Hours,Rate,Name$)     ! 3 parameters

450  SUB X(INTEGER D,C(*),B$,A)      ! D & C(*) are integers;
                                       A is real-precision
451                                  ! since it is after a string

570  SUB Store_data(A,B,C,#4)        ! Can pass file numbers

900  SUB Get_data(SHORT B(*),#7,A)   ! A is real-precision since
                                       it is after a file number
```

Type words are cumulative like in a COM statement. For example, if INTEGER is specified, all variables following it are declared as being integers until a string, a file number or another type word is specified.

Pass Parameters

The pass parameter list is used in calling the subprogram (using CALL or FN) and includes numeric and string variable names, array identifiers, numeric expressions and file numbers in the form: # file number. Parameters must be separated by commas. The pass parameter list must also be enclosed in parentheses.

All array variables in the pass parameter list must be defined within the calling program. That is, arrays must have been dimensioned, either implicitly or explicitly.

Passing the Parameters

When a subprogram is called, (with CALL or FN) each formal parameter is associated with and assigned the value of the pass parameter which is in the corresponding position in the pass parameter list. The parameter lists must have the same number of parameters; the parameters must match in type – numeric or string, simple or array.

Example

The following example shows a formal parameter list, (SUB, line 300) and two corresponding pass parameter lists (CALL, lines 70 and 150).

```
  10   DIM A(3,3)
  20   INTEGER C(2,2),D(2,2)
• 70   CALL X(A(2,3),B$,C(*),E+F,#6)
  75   !
• 150  CALL X(5,(C$[1,12]),D(*),M,#2)
  290  END
• 300  SUB X(X,Y$,INTEGER Z(*),REAL K,#3)
  305  !
  310  SUBEND
```

Notice the correspondence between pass and formal parameters. Notice also that the arrays C and D are dimensioned (line 20) before being passed.

Parameters are passed either by **reference** or by **value**. When a parameter is passed by reference, the corresponding formal parameter shares the same memory area with the pass parameter. Thus, changing the value of the corresponding variable in the subprogram changes the corresponding value of the variable in the calling program.

When a parameter is passed by value, the variable defined by the corresponding formal parameter is assigned the value of the pass parameter and given its own temporary storage space in memory. Numeric and string expressions are necessarily passed by value. However, arrays can't be passed by value. Enclosing a pass parameter in parentheses causes it to be considered an expression and thus passed by value, rather than by reference. Passing by value prevents the value of a calling program variable from being changed within a subprogram.

Examples

In the following example all parameters in line 80 are passed by value; those in line 130 are passed by reference.

```
  10   DIM L$[25]
  20   L$="GIVE THE STRING A VALUE"
  25   !
  75   ! BY VALUE:
• 80   CALL Active(Y+3,(X(1,4)),PI,(Y),(L$[10]))
  81   !
  85   ! BY REFERENCE:
• 130  CALL Active(Y,X(1,4),A,Z,L$)
  135  !
  175  END
  180  SUB Active(A,B,C,D,F$)
  240  SUBEND
```

Here is an example of similar program segments. Notice the value of X in each case.

Pass by value

```
  10   A=1
  20   PRINT "A before pass by value:";A
● 30   CALL X((A))        ! Pass by value
  40   PRINT "A after pass by value:";A
  50   END
  60   SUB X(P)
  70   P=P+P
  80   SUBEND
```

```
A before pass by value: 1
A after pass by value: 1
```

Pass by reference

```
  10   A=1
  20   PRINT "A before pass by reference:";A
● 30   CALL X(A)
  40   PRINT "A after pass by reference:";A
  50   END
  60   SUB X(P)
  70   P=P+P          ! Changes value of A in calling program
  80   SUBEND
```

```
A before pass by reference: 1
A after pass by reference: 2
```

Any parameters passed by value are converted, if necessary, to the numeric type – REAL, SHORT, INTEGER – of the corresponding parameter in the formal parameter list. For example, say that PI is passed by value to an INTEGER formal parameter. Its value would be rounded to 3 when the subprogram is called.

Those passed by reference must match exactly, otherwise ERROR 8 occurs and no conversion is made.

Computer Museum

Summary

Here are some facts to remember concerning parameters.

- Formal parameters are used in defining the subprogram (in the DEF FN or SUB statement) and can be simple variables, array identifiers or file numbers.
- Pass parameters are used in the calling program (FN or CALL statement) to pass values to the subprogram and can be single variables, array identifiers, expressions or file numbers.
- The parameter list must be enclosed in parentheses and all parameters must be separated by commas.
- Numeric type – INTEGER, SHORT and REAL – can be declared in the formal parameter list.
- Parameters can be passed by reference or by value. Enclosing a pass parameter in parentheses causes it to be passed by value. Parameters passed by reference must match in numeric type. Numeric and string expressions are always passed by value, while arrays are always passed by reference.

Multiple-Line Function Subprograms – DEF FN

The multiple-line function subprogram is used to define a numeric or string function which returns a value (numeric or string) to the calling program. There are four syntax which are used with multiple-line function subprograms –

- DEF FN subprogram name [‹formal parameter list›]
  DEF FN subprogram name $ [‹formal parameter list›]

  The DEF FN statement is the first line of a user-defined multiple-line function subprogram. The second syntax is used for defining a string function. The subprogram name must be a valid name.

- FN END

  The FN END statement is the last statement in a multiple-line function subprogram.

- RETURN numeric expression
  RETURN string expression

  The RETURN statement specifies the value (numeric or string) which is to be returned to the calling program for the value of the function. RETURN also transfers control back to the calling program.

- FN subprogram name [‹pass parameter list›]
  FN subprogram name $ [‹pass parameter list›]

  FN is used to reference the subprogram. When it is encountered, values are passed and control is transferred to the subprogram. FN references to multiple-line functions can't appear in an input or output statement or in redim subscripts.

Examples

Here's an example of a numeric function –

```
  10   DIM C(2,2)              ! Must dimension arrays to pass them
  20   A=RND
  30   B=7
  40   MAT C=(8*RND)
• 50   Y=FNTotal(A,B,C(*))
  60   PRINT "TOTAL IS";Y      ! FNTotal can't be in PRINT statement
  70   END
• 80   DEF FNTotal(X,Y,Z(*))! DEF FN is first line in a multiple-
                               line function subprogram
• 90   RETURN SUM(Z)+X+Y       ! Provides the value
• 100  FNEND                   ! FNEND is last line in a multiple-
                               line function subprogram
```

```
TOTAL IS 35.1956728609
```

Here's an example of a string function –

```
10    DIM Name$[100],Job$[100],A$[100]
20    DATA J. SMITH, ENGINEER,7,B. JONES,BANJO PLAYER,9
30    FOR I=1 TO 2
40       READ Name$,Job$,Level
50       A$=FNClassify$(Name$,Job$,Level)
60       PRINT A$
70    NEXT I
80    END
90    DEF FNClassify$(X$,Y$,Z)
100   RETURN X$&"'S JOB IS "&Y$&" WHICH IS ON LEVEL "&VAL$(Z)
110   FNEND
```

```
J. SMITH'S JOB IS ENGINEER WHICH IS ON LEVEL 7
B. JONES'S JOB IS BANJO PLAYER WHICH IS ON LEVEL 9
```

There can be more than one RETURN statement in a subprogram, but only one is executed each time the subprogram is executed.

Example

```
10    Reg_hours=40
20    Rate=4.75
30    Ovrtm_rate=Rate*1.5
40 Extra_hours:  DATA 5,3,0,6
50    FOR I=1 TO 4
60       READ Ovrtm
70       Wages=FNPay(Reg_hours,Rate,Ovrtm_rate,Ovrtm)
80       PRINT Ovrtm;"HOURS OVERTIME, PAY IS";Wages
90    NEXT I
100   END
110   DEF FNPay(A,B,C,D)
120   Pay=A*B
130   IF D<=0 THEN RETURN Pay
140   !  There can be more than one RETURN, but only one is
         executed each time the subprogram is accessed
150   RETURN Pay+C*D
160   FNEND
```

```
 5 HOURS OVERTIME, PAY IS 225.625
 3 HOURS OVERTIME, PAY IS 211.375
 0 HOURS OVERTIME, PAY IS 190
 6 HOURS OVERTIME, PAY IS 232.75
```

If a single-line and multiple-line function are defined with the same name and that name is referenced, the single-line function is accessed if it is defined within that program segment.

Subroutine Subprograms – SUB and CALL

Subroutine subprograms allow you to repeat a series of operations many times using different values or to break a large problem down into a series of smaller ones. A subroutine subprogram performs a specific task.

There are four statements which are used with subroutine subprograms –

- SUB subprogram name [⟨formal parameter list⟩]

  The SUB statement is the first statement of a subroutine subprogram. The subprogram name must be a valid name.

- SUBEND

  The SUBEND statement is the last line of a subroutine subprogram and transfers control back to the calling program.

- SUBEXIT

  The SUBEXIT statement can be used within the body of a subprogram to transfer control back to the calling program before SUBEND is executed.

- CALL subprogram name [⟨pass parameter list⟩]

  The CALL statement is used to transfer control and pass values to the subprogram.

Examples

Here is a simple example of a subroutine used to write a heading for data output. Notice that no parameters are passed.

```
• 10   CALL Heading
  20   END
• 30   SUB Heading    ! SUB is 1st line of subroutine subprogram
  40   PRINT TAB(11),"NAME";TAB(27),"AMOUNT"
  50   PRINT RPT$("_",40)
• 60   SUBEND         ! SUBEND is last line of subroutine subprogram
                        it also returns control to calling program
```

```
          NAME                AMOUNT
_________________________________________
```

Here is another example which manipulates the parameters and could be used to output a readable table supplied with a value for N (line 30) –

```
10   OPTION BASE 1
20   DIM Prodnums(7),Percent(7)
30   N=5                        ! NUMBER OF PRODUCTS FOR THIS TRY
40 Prodnums:      DATA 25,15,31,30,45,97,35
50 Percents:      DATA 27,13,10,12,18,2,28
60   MAT READ Prodnums(N)   ! REDIMS Prodnums TO # OF PRODUCTS
70   RESTORE Percents       ! READ DATA FROM Percents
80   MAT READ Percent(N)
90   CALL Table(Prodnums(*),Percent(*),N)
100  END
110  SUB Table(Pr(*),Per(*),N)
120  OPTION BASE 1          ! OPTION BASE VALUES MUST MATCH
130  PRINT "PRODUCT","% OF SALES"
140  FOR I=1 TO N           ! ONE LINE PER PRODUCT
150     PRINT Pr(I),Per(I)
160  NEXT I
170  SUBEND
```

```
PRODUCT                % OF SALES
 25                     27
 15                     13
 31                     10
 30                     12
 45                     18
```

The SUBEXIT statement is used to transfer control back to the calling program before SUBEND is executed.

Example

```
10   INPUT "VALUES FOR Rate AND Hours",R,H
20   PRINT "RATE:";R,"HOURS:";H
30   CALL Pay(R,H)
40   END
50   SUB Pay(X,Y)
60      Pay=X*Y
70      IF Y>40 THEN Bonus
80      PRINT "PAY IS:";Pay
90      SUBEXIT        ! Control returns to calling program
100 Bonus:  Extra_pay=Pay*.25
110     PRINT "PAY WITH BONUS IS :";Pay+Extra_pay
120     SUBEND
```

```
RATE: 5.5            HOURS: 45
PAY WITH BONUS IS : 309.375
```

Subprogram Considerations

Entering a Subprogram

When a subprogram is entered the following occur –

- The DATA pointer is reset to the first DATA statement in subprogram.
- Any file assignments that are not passed are cleared.
- RAD, STANDARD, and OPTION BASE 0 are the modes defaulted to.
- Any ON KEY#, ON KBD, ON END#, ON INT or ON ERROR associated with a GOTO or GOSUB is no longer active; however one ON KEY interrupt per key and up to 80 ON KBD keystrokes are logged for processing upon return to the calling program. Interrupts associated with CALL remain active.

Upon return to the calling program, all of these conditions are restored to their previous state.

Using the COM Statement

Values can also be passed to a subprogram with a COM statement. The list of items in the subprogram COM may be a subset of the main program COM statement; that is, it must match up to some point in the main program COM. A variable can't be an item in a subprogram COM statement if it is also in the formal parameter list.

Example

```
  10   OPTION BASE 1
• 20   COM A(4,4),B,INTEGER C,D(3,3),E$[28]
  30   MAT A=(5)
  40   B=C=7
  50   MAT D=(3)
  60   CALL Routine
  70   Y=FNRandom
  80   PRINT Y
  90   END
  100  SUB Routine
  110      OPTION BASE 1          ! OPTION BASE value must match if
                                    any arrays passed in COM list
• 120      COM X(4,4),Y,INTEGER Z,Q(3,3)
  130      PRINT X(*);Y,Z,LIN(2),Q(*);
  140      SUBEND
  150  DEF FNRandom
• 160      COM I(1:4,1:4)         ! The lower bound can be specified
                                    to make OPTION BASE value match
  170      RETURN I(2,2)*RND
  180      FNEND
```

```
 5  5  5  5

 5  5  5  5

 5  5  5  5

 5  5  5  5

 7                    7

 3  3  3

 3  3  3

 3  3  3

 3.39109504673
```

Arrays can be specified in a subprogram COM statement using an array identifier. This method is useful for editing in that if you change the dimensions of an array in a main program COM statement, you won't have to edit each subprogram COM to make the dimensionality match. Using an array identifier also avoids an error if an array declared with COM was redimensioned in the calling program segment.

Example

```
  10    OPTION BASE 1
• 20    COM Array(4,4),Value
  30    Value=2
  40    MAT Array=(5)
• 50    REDIM Array(2,2)
  60    CALL Sub
  70    END
  80    SUB Sub
  90        OPTION BASE 1
• 100       COM A(*),B      ! Array identifier is a good way to pass
                              an array.  The REDIM  doesn't matter
  110       MAT A=A*(B)
  120       MAT PRINT A
  130       SUBEND
```

```
 10                   10

 10                   10
```

Variable Allocation Statements

Subprograms may also have any variable allocation statements: DIM, REAL, SHORT, and INTEGER. However, the variables declared may not be in the subprogram COM statement or the formal parameter list.

Example

```
  10   CALL X(5,3)
  20   END
  30   SUB X(A,B)
• 40      DIM F(2,4),G$(2,2)[50]    ! Can use expressions
• 50      SHORT H(A,B)
  60   !    Rest of subprogram
  70      SUBEND
```

Within subprogram variable allocation statements, array subscripts and maximum string lengths can be specified with numeric expressions that can contain both constants and formal parameters.

Local Variables

All variables in a subprogram that are not part of the formal parameter list or the COM statement are known as "local" variables and cannot be accessed from any other program segment. Storage of local variables is temporary, and is returned to main user Read/Write Memory upon return to the calling program. This is known as **dynamic memory allocation**.

All variable names in a subprogram are independent of variables with the same name in other program segments. Thus, if you check the value of a variable using live keyboard while a program is running, you may get an unexpected result if the variable is defined differently in the program segment which is executing currently.

Speed Considerations

CALLs to subprograms cause program execution to be slower than if GOSUBs are used. Thus, in situations where the separate environment of a subprogram is not needed, it is advantageous to use GOSUB and a subroutine instead.

Files

File numbers of files opened in the calling program can be passed to a subprogram in the parameter list.

Example

```
 10   CREATE "Data",1
● 20   ASSIGN #1 TO "Data"
● 30   CALL Routine(#1)      ! File assignment is passed
 40   READ #1,1;A           ! PRINT# is in subprogram
 50   PRINT A
 60   END
● 70   SUB Routine(#3)       ! #3 is now assigned to "Data"
 80      PRINT #3;RND
 90      SUBEND
```

```
 .678219009345
```

Any operations, such as PRINT#, which involve file #3 in the subprogram will affect file #1, Data, in the calling program.

File numbers can also be implicitly assigned within the calling program from within a subprogram.

Example

```
 10   CREATE "Pay",1
● 20   CALL X(#4)
 30   READ #4,1;A        ! No ASSIGN in calling program
 40   PRINT A
 50   END
● 60   SUB X(#2)
● 70      ASSIGN #2 TO "Pay"
 80      PRINT #2,1;RND
 90      SUBEND
```

```
 .678219009345
```

When control returns to the calling program, #4 is still assigned to the file Pay.

A file can also be implicity buffered in this manner.

Example

```
  100  CALL Data(#4)
  310  END
  320  SUB Data(#2)
  330  ASSIGN #2 TO "Pay"
• 340  BUFFER #2      ! Implicitly buffers #4 in calling program
  350  SUBEND
```

When control returns to the calling program, #4 is still assigned to Pay and it is still buffered.

If a file is actually opened in a subprogram and wasn't passed as a parameter, it is automatically closed upon return to the calling program.

Editing Subprograms

There are two ways to add a new subprogram to a main program and any subprograms. It must either replace an existing subprogram or it must come **after** all other subprograms. You can't insert a subprogram between the last line of one subprogram and the first line of the next. Using a mass storage device and storing parts of a program can allow you to get around this and insert a subprogram.

In order to delete the first line of a subprogram (the SUB or DEF FN statement), the entire subprogram must be deleted. You can't combine two subprograms by deleting the SUB or CALL of the second one.

The SUB statement can be edited as long as it remains a SUB statement or is changed to a DEF FN.

Chapter 10
Output

page 144 • **BEEP** (outputs an audible tone)
page 144 • **DISP** (outputs text and variables to the display line of the CRT)
page 146 • **PRINTER IS** (defines the standard printer for PRINT, PRINT USING, LIST and CAT operations)
page 147 • **PRINT** (outputs text and variables to the standard printer)
page 153 • **MAT PRINT** (prints entire arrays on the standard printer)
page 156 • **PRINT USING, IMAGE** (let you format printed output exactly like you want it)
page 167 • **OVERLAP** (sets overlapped processing mode)
page 167 • **SERIAL** (sets serial processing mode)

Output Functions

TAB (tab to column – DISP and PRINT)
SPA (skip spaces – DISP and PRINT)
LIN (output linefeeds – PRINT)
PAGE (go to the next page – PRINT)

Terms

- Standard printer – the printing device to which output from PRINT, PRINT USING, LIST and CAT is directed. At power on and SCRATCH A, it is the CRT.
- Select code – an expression in the range 0 through 16 used to access an input or output device. The following select codes are reserved –
  0 – Internal printer and keyboard
  13 – Graphics option
  14 – Optional tape drive
  15 – Standard tape drive
  16 – CRT
- Format string – specifies the format for PRINT USING output
- Overlapped processing mode – allows computation and I/O to run simultaneously.
- Serial processing mode – computation and I/O statements do not run simultaneously, but are executed one at a time.

Spacing

, – causes the item it follows in a PRINT or DISP list to be output in a 20-character field.

; – causes the item it follows in a PRINT or DISP list to be output with no additional blanks.

The BEEP Statement

The BEEP statement is used to create a brief audible tone which can be used in a number of ways.

10

```
BEEP
```

BEEP can signal that a particular computation or program segment is complete. It can also be used to indicate audibly that the computer is ready for input, so that the operator does not have to remain at the keyboard.

Example

Here's an example use for BEEP –

```
10   DIM N(1:7)
20   FOR I=1 TO 7
30      BEEP          ! Signals user when an input is required
40      INPUT "DATA VALUE?",N(I)
50   NEXT I
60   PRINT N(*)
70   END
```

In this case, a beep signals the operator when the program is ready for input.

The DISP Statement

The DISP (display) statement allows text and variables to be output in the display line.

DISP [display list]

The display list can contain the following –

- variable names
- array identifiers
- numeric expressions
- string expressions
- TAB function[1]
- SPA function[1]

Multiple-line user-defined functions aren't allowed in the display list, alone or in an expression. Items in the display list must be separated by commas or semicolons. The list may end with a comma or semicolon, which causes the next display to be appended to the display line. Otherwise, one display replaces the previous one.

1 The output functions are discussed later in this chapter.

Computer Museum

Example

```
  10    X=3.5
• 20    DISP "X EQUALS";X,"X SQUARED EQUALS";X^2
  30    !     Note the use of commas and semicolons
  40    END

  X EQUALS 3.5          X SQUARED EQUALS 12.25
```

Notice the difference in spacing between the numbers caused by the use of a comma or a semicolon. When an item is followed by a comma, it is left justified in a field that is 20 characters wide. Two or more commas after an item cause one or more character fields to be skipped. When an item is followed by a semicolon, no additional blanks are output after the item. Remember that every number has a leading blank or minus sign and a trailing blank.

Examples

```
• 10    DISP 125000,,250000       ! 2 commas separate items
  20    END

   125000                                   250000
```

```
• 10    DISP 100;-20;77.3
  20    END

   100 -20  77.3
```

```
• 10    DISP 10,20,30,40          ! 20-character fields
  20    WAIT 2000                 ! Lets both displays be seen
• 30    DISP 10;20;30;40          ! Tight spacing
  40    END
```

The following lines are displayed two seconds apart.

```
   10                   20                   30                   40
   10  20  30  40
```

```
10   Date$="June 1"
20   DISP "Today is ";      ! More displays are appended
30   WAIT 1000
40   DISP Date$
50   END
```

The following is displayed –

```
Today is
```

and then changes to –

```
Today is June 1
```

If the information being displayed is longer than 80 characters, a carriage return/linefeed (CR-LF) is automatically output after every 80th character causing a new line to overwrite the previous one. Only the last line of the displayed information is visible. You can see all of the displayed information by setting the print all mode (press and latch PRT ALL). This causes every display to be printed on the print all printer.

Printed Output

Five statements are used to control printed output: PRINTER IS, PRINT, MAT PRINT, PRINT USING, and IMAGE.

The PRINTER IS Statement

The PRINTER IS statement defines the standard print device for the system. The CRT, select code 16, is standard at power on, and SCRATCH A.

> PRINTER IS select code [, HP-IB device address] [, WIDTH number of characters per line]

All output from PRINT, PRINT USING, LIST and CAT, and syntax error messages from GET or LINK are directed to the standard printer.

The specified device must be an acceptable printing device, like a printer or tape punch; it may be any device which can accept strings of ASCII characters.

The WIDTH parameter is a numeric expression and specifies the number of characters per line of the standard printer. This determines when a carriage return-linefeed will be output. One is output when the number of characters printed equals the width of the line. Its range is 16 through 260; 80 is the power on and default value when one isn't specified.

Examples

```
10   !  PRINTER IS examples;  don't run this
20   PRINTER IS 16              ! Printer is CRT
30   PRINTER IS 0               ! Printer is internal printer
40   PRINTER IS 6               ! Printer at select code 6
50   PRINTER IS 6,WIDTH(160)    ! 160 characters per line
60   PRINTER IS 7,2             ! HP-IB printer
70   END
```

The PRINT Statement

The PRINT statement causes text and variables to be output on the standard printer.

PRINT [print list]

The print list can contain the following items –

- variable names
- array identifiers
- numeric expressions
- string expressions
- TAB function
- SPA function
- LIN function
- PAGE function

Multiple-line user-defined functions aren't allowed, alone or in an expression. All items must be separated by commas or semicolons.

Examples

```
  10   PRINTER IS 16
  20   FOR I=1 TO 5
• 30      PRINT "I EQUALS";I
  40   NEXT I
• 50   PRINTER IS 16,WIDTH(40)   ! CHANGE WIDTH
• 60   PRINT RPT$("X",100)       ! PRINTS 100 X's
  70   END
```

```
I EQUALS 1
I EQUALS 2
I EQUALS 3
I EQUALS 4
I EQUALS 5
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
```

```
● 10   PRINT "***!!!";"///^^^";"%%%@@@"   ! Tight spacing
● 20   PRINT "***!!!","///^^^","%%%@@@"   ! 20-character fields
  30   END

  ***!!!///^^^%%%@@@
  ***!!!                  ///^^^                  %%%@@@
```

Notice in the previous example that commas and semicolons cause the same spacing in the PRINT statement as they do in the DISP statement. A comma after an item causes it to be left justified within a 20-character field. A semicolon after an item suppresses any additional blanks other than the leading blank or sign and the trailing blank. A comma or semicolon after the last item in the list allows a future print list to be appended by suppressing the CR-LF. A CR-LF is automatically output when the WIDTH is exceeded.

The current numeric output form (STANDARD, FIXED or FLOAT) determines how a number is output with both DISP and PRINT.

Example

```
● 10   FIXED 2
  20   GOSUB Print
  30   FLOAT 3
  40   GOSUB Print
  50   STOP
● 60 Print:  PRINT 20;81.1596;32.9
  70   RETURN
  80   END

  20.00  81.16  32.90
  2.000E+01  8.116E+01  3.290E+01
```

The variable width of the standard printer can be especially useful when outputting non-printable characters such as escape codes. Although you can't see a non-printable character, the computer counts it in when it is keeping track of how many characters it has printed to a line and you may get a carriage return-linefeed before the line is filled with printed characters.

Example

Here is an example to try which uses the CRT as the output device –

```
10   !   *****   TRY this example   *****
20   PRINTER IS 16,WIDTH(160)     ! Normal width is 80
30   FOR I=1 TO 40
40      PRINT CHR$(129)&"*"&CHR$(128)&"*";
        ! CHR$(128) and CHR$(129) access CRT special features
50      ! Line 40 prints 4 characters at a time--160 total
60   NEXT I
70   END
```

In this example, CHR$(129) and CHR$(128) are non-printable characters used to turn inverse video mode on and off. Please refer to Appendix A for more explanation of this use of CHR$.

Output Functions

Four output functions are available to increase formatting capabilities. TAB and SPA can be used with both DISP and PRINT; LIN and PAGE can be used only with PRINT. They must be separated from the next item in the display or print list with either a comma or a semicolon. However, both the comma and semicolon function identically after an output function; they merely serve to separate it from the next item.

The TAB Function

The TAB function causes the next item in the list to be output beginning in the specified column.

TAB character position

The character position can be specified by any numeric expression, except one containing a multiple-line function, and it is rounded to an integer. If it is less than 1, it defaults to 1.

Example

```
10   PRINT 147;TAB(10),"THIS STARTS IN THE 10TH COLUMN"
20   ! Notice the semicolon after 147
30   END
```

```
 147      THIS STARTS IN THE 10TH COLUMN
```

If the specified column has already been filled, a CR-LF is output, and then the TAB is completed.

Example

If the PRINT statement in the previous example is changed to –

```
• 10   PRINT 147,TAB(10),"THIS STARTS IN THE 10TH COLUMN"
  20   ! Notice the comma after 147
  30   END
```

```
 147
          THIS STARTS IN THE 10TH COLUMN
```

a CR-LF would be output after 147, since the comma causes 147 to be output in a 20-column field, then the tab occurs.

When the character position specified is greater than the number of columns in the standard printer, it is reduced by this formula –

(character position – 1) MOD N + 1

N is the number of columns specified as standard printer width.

Example

For example, with printer width 80 –

```
10 PRINT 0; TAB (10), 1; TAB (90),2;TAB (170),3
```

```
 0         1
           2
           3
```

If you are printing non-printable characters and using TAB, you may get unexpected results if you haven't taken the non-printable characters into consideration.

The SPA Function

The SPA (space) function is used with DISP and PRINT to output the specified number of blank spaces up to the end of the current line.

SPA number of spaces

Computer Museum

Example

```
DISP 1; SPA (10);2;SPA(10),3

 1          2          3
```

The number of spaces can be specified by any non-negative numeric expression, except one containing multiple-line function, and it is rounded to an integer. If it specifies more blanks than remain in the line, the next item begins the next line.

Example

```
10 PRINT "**"; TAB 70,"**"; SPA 20, "**"
```

is printed –

```
**                                                                    **
**
```

The LIN Function

The LIN function is used with PRINT and causes the specified number of linefeeds to be output.

LIN number of linefeeds

The number of linefeeds can be specified by any numeric expression, except one containing a multiple-line function, and it is rounded to an integer. Its range is −32 768 through 32 767.

Example

```
 10   A$="AUGUST 28"
● 20   PRINT "TODAY IS ";A$,LIN(3),"DATA COMPLETE"
 30   !      A carriage return and 3 linefeeds are
            output between strings
 40   END


TODAY IS AUGUST 28


DATA COMPLETE
```

When the number of linefeeds is positive, a carriage return precedes the linefeeds. When zero linefeeds are specified, only a carriage return is output. This can cause some interesting printing on the internal thermal printer because the paper is backed up and the second line is printed on top of the first. When the number of linefeeds is negative, no carriage return is output; the number of linefeeds output equals the absolute value of the expression. Some external printers can't suppress the carriage return.

Example

```
10    PRINTER IS 0
20    PRINT "Today";LIN(-2);"Is";LIN(-2),"Saturday"
30    END
```

```
Today

    Is

        Saturday
```

The PAGE Function

The PAGE function can be used with PRINT and causes a form feed character to be output, so further printing can begin on a new page or at the top of the next form on devices that can understand ASCII form feed (CHR$(12)). The formfeed action varies from device to device. When the standard printer is the CRT, PAGE clears the entire print area.

```
PAGE
```

Example

In this example, 'RESULT' and B(*) are printed on a new page.

```
100 PRINT"DATA";LIN(2),A(*),PAGE,"RESULT"; LIN(2);B(*)
```

The MAT PRINT Statement

The MAT PRINT statement is used to print arrays on the standard printer.

MAT PRINT array variable [, or ; [array variable, or ;...]]

10

The comma or semicolon following an item specifies open or close spacing between the elements.

Example

```
  10    OPTION BASE 1
  20    DIM A(3,3)
  30    MAT A=(5)
  40    PRINT "20-CHARACTER FIELDS"
● 50    MAT PRINT A
  60    PRINT
  70    PRINT "CLOSE SPACING"
● 80    MAT PRINT A;          ! Notice the semicolon's
                                effect
  90    END
```

```
20-CHARACTER FIELDS
 5                   5                   5

 5                   5                   5

 5                   5                   5


CLOSE SPACING
 5  5  5

 5  5  5

 5  5  5
```

When an array is printed, every printed row is followed by a blank line. The last row is followed by two blank lines.

When an array has more than two dimensions, the last subscript varies fastest and defines the length of a row.

10

Example

```
10   OPTION BASE 1
20   DIM A(2,3,4)
30   FOR I=1 TO 2            ! Affects 1st (left) subscript
40      FOR J=1 TO 3         ! Affects 2nd subscript
50         FOR K=1 TO 4      ! Affects 3rd (right) subscripts
60            A(I,J,K)=X     ! Assigns values to array elements
70            X=X+1
80         NEXT K
90      NEXT J
100  NEXT I
110  PRINT "THREE-DIMENSIONAL ARRAY - (2,3,4)"
120  MAT PRINT A;
130  END
```

```
THREE-DIMENSIONAL ARRAY - (2,3,4)
 0  1  2  3

 4  5  6  7

 8  9  10  11

 12  13  14  15

 16  17  18  19

 20  21  22  23
```

In this example, array A(2,3,4) is interpreted as two matrices, each 3 by 4, for output or input purposes.

Arrays can also be printed by the PRINT statement using an array identifier, (*). In the previous example, line 120 could be changed to –

```
120   PRINT A(*);
```

The PRINT USING and IMAGE Statements

Two statements, PRINT USING and IMAGE, provide the capability of generating printed output with complete control of the format. This is done by referencing a list of specifiers called a **format string**. The format string can be listed in an IMAGE statement, then used by referencing its line identifier in a PRINT USING statement. Or, the format string can be contained in a string expression which is used in place of the line identifier in the PRINT USING statement.

PRINT USING string expression[; print using list]

PRINT USING line identifier[; print using list]

IMAGE format string

The print using list can contain the following items –

variable names
array identifiers
numeric expressions
string expressions

No multiple-line user-defined functions can be specified in the print using list. The items in the list are separated by commas or semicolons. However, the commas and semicolons have no effect on the printout, as in PRINT or DISP; they are used only to separate items. The output is totally controlled by the format string.

The string expression in the first syntax must be a valid format string at the time of execution. It can be any string expression. The line identifier in the second syntax must refer to an IMAGE statement that contains the format string corresponding to the particular PRINT USING statement.

Format String

The format string is a list of **field specifiers** separated by delimiters. It is used to specify numeric and string fields, blanks, and carriage control. Each numeric or string field specifier must correspond to an appropriate item in the print using list. Each field specifier is made up of various symbols and determines how a single item in the print using list is to be output.

Reusing the Format String

A format string is reused from the beginning if it is exhausted before the print using list. This is also a way to replicate fields.

Example

```
10   IMAGE DDD.DD                  ! One numeric field
20   PRINT USING 10;25.71,99.9     ! Two numbers, image reused
30   END
```

```
 25.71 99.90
```

Delimiters

Three delimiters are used to separate field specifiers –

, A comma is used only to separate two specifiers.

/ A slash can be used to separate two specifiers. It also causes output of a CR-LF.

@ The commercial-at sign can be used to separate two specifiers. It also causes output of a formfeed character, starting a new page of output on devices that have this capability.

The / and @ symbols can also be used as field specifiers by themselves; that is, they may be separated from other specifiers by a comma. Only the / can be directly replicated. You could output three CR-LF's with /// or 3/

Blank Spaces

A blank space is specified with –

X NX specifies N blanks. Any X specifier can be embedded within any other field specifier without delimiters.

String Specification

Text can be specified in two ways –

" " A literal specifier is text enclosed in quotes. This specifier may be embedded without delimiters within any other field specifier.

A A is used to specify a single string character. NA specifies N characters. The length of the string specifier is determined by the number of A's that are specified between delimiters; this corresponds to one item in the print using list.

Example

```
10   IMAGE "**",4X"Results"4X,"**"    ! Blanks and literals output
20   PRINT USING 10
30   END
```

```
**    Results     **
```

This example can also be written –

```
10   A$="Results"
20   IMAGE "**"4X7A4X"**"    ! Literal, blank & string specified
30   PRINT USING 20;A$
40   END
```

```
**    Results     **
```

If the string item in the print using list is longer than the number of characters specified, the string is truncated.

Example

```
10   PRINT USING "5A";"RESULTS"
20   END
```

```
RESUL
```

If the item is shorter, the item is left justified and the rest of the field is filled with blanks.

Numeric Specification

Numeric field specifiers can be made up of various types of symbols: digit symbols, sign symbols, radix symbols, separator symbols and an exponent symbol. These are covered next.

Digit Symbols

D Specifies a digit position. ND specifies N digit positions. Leading zeros are replaced with a blank space as a fill character.

Example

```
10    PRINT USING "DDDDD,2X,DD";250,25
20    !        Line 10 specifies a 5-digit field,
               2 blanks and a 2-digit field
30    END

250  25
```

Z Specifies a digit position. NZ specifies N digit positions. Leading zeros are replaced with 0 as a fill character.

Example

```
10    IMAGE XAXXXX,AAA/              ! Two literal fields & blanks
                                       followed by CR/LF
20    IMAGE ZZZ,3X,ZZZ               ! Two 3-character fields
30    PRINT USING 10;"I","I*4"       ! Print heading
40    FOR I=1 TO 3
50       PRINT USING 20;I,I*4        ! Print 2 numbers
60    NEXT I
70    END

 I     I*4

001    004
002    008
003    012
```

---

* Specifies a digit position. N* specifies N digit positions. Leading zeros are replaced with * as a fill character.

Example

```
10    IMAGE *****,2X,"Dollars and",XDDX,"cents"//
20    !     5 digits, 2 blanks, literal, digits & blanks,
30    !     literal and 2 CR/LF's
40    PRINT USING 10;250,52   ! Dollars & cents are separate
50    PRINT USING "36A";"(*'s are good for check protection)"
60    END

**250  Dollars and 52 cents


(*'s are good for check protection)
```

Only the symbol D is allowed to the right of any radix indicator symbol (discussed next). Any digit symbol can be used to specify the integer portion of any number but, with one exception, they can not be mixed. That is, for example, if D is used they must all be D. The exception is that the digit symbol specifying the one's place can be a Z regardless of the other symbols.

Example

```
10    IMAGE DDD.DD/***.DD       ! D's and *'s to left of radix
20    IMAGE DDZ.DD/**Z.DD       ! Like above but Z in 1's place
30    PRINT USING 10;.25,.75
40    PRINT USING 20;.25,.75
50    END
```

```
   .25
***.75
  0.25
**0.75
```

Radix Symbols

A radix indicator is used to separate the integer part of a number from the fractional part. In the United States for example, this is customarily the decimal point, as in 34.7. In Europe, this is frequently the comma as in 34,7. Only one symbol for a radix indicator, at most, can appear in a numeric specifier.

. Specifies a decimal point radix indicator in that position.

R Specifies a comma radix indicator in that position.

Examples

```
10    A$="United States"
20    E$="Europe"
30    IMAGE DDD.DD,2X,13A        ! Decimal point as radix
40    IMAGE DDDRDD,2X,6A         ! Comma as radix
50    PRINT USING 30;225.05,A$
60    PRINT USING 40;225.05,E$   ! Radix must be a period in
                                   the print using list
70    END
```

```
225.05  United States
225,05  Europe
```

If the number to be output contains more digits to the right of the radix indicator than are specified, the number is rounded.

Example

```
10    IMAGE DD.DD              ! 2 places after decimal
20    PRINT USING 10;25.256    ! 3 places after decimal
30    END
```

```
25.26
```

Sign Symbols

Two sign symbols are used to control the output of the sign characters + and −. Only one sign symbol at most can appear in a numeric specifier.

S Specifies output of a sign: + if the number is positive, − if the number is negative.

M Specifies output of a sign: − if the number is negative, a blank if it is positive.

If the sign symbol appears before all digit symbols in a numeric specifier, it floats (see the section on Floating Symbols which is later in this chapter) to the left of the leftmost significant digit output.

When no sign symbol is specified and the number to be output is negative, the minus sign occupies a digit position.

Example

```
10    IMAGE SDDD,3X,MDDD       ! One with S, one with M
20    PRINT USING 10;250,250   ! Both numbers positive
30    PRINT USING 10;-5,-10    ! Both numbers negative
40    IMAGE /SDDDD.DD,2X,"Monthly profit"
50                             ! Sign floats
60    PRINT USING 40;25.15,-4000.25
70    END
```

```
+250     250
  -5     -10

  +25.15  Monthly profit
-4000.25  Monthly profit
```

Digit Separator Symbols

Digit separators are used to break large numbers into groups of digits (generally three digits per group) for greater readability. In the United States, the comma is customarily used; in Europe, the period is commonly used. The X symbol can also be used to cause digits to be separated with a blank space.

C Specifies a comma as a separator in the specified position.

P Specifies a period as a separator in the specified position.

The digit separator is output in an item only if a digit in that item has already been output; the separator must appear between two digits. When leading zeroes are generated by the Z symbol, they are considered digits and will contain separators if specified.

Example

```
10   PRINT "DIGIT SEPARATOR NOT OUTPUT IN 2nd NUMBER:"
20   IMAGE DDDCDDD,2X,DDDCDDD/
30   PRINT USING 20;2525,250   ! No comma output in second number
40   !
50   PRINT "EUROPEAN VS. AMERICAN:"
60   IMAGE DDDPDDDPDDD,2X,"Houses in Hamburg"
70   IMAGE DDDCDDDCDDD,2X,"Houses in Loveland"/
80   PRINT USING 60;21345
90   PRINT USING 70;19874
100  !
110  PRINT "USING BLANKS TO SEPARATE DIGITS:"
120  PRINT USING "Z.5DX5DX2D";SIN(1) ! Blanks separate digits
130  END
```

```
DIGIT SEPARATOR NOT OUTPUT IN 2nd NUMBER:
  2,525      250

EUROPEAN VS. AMERICAN:
     21.345  Houses in Hamburg
     19,874  Houses in Loveland

USING BLANKS TO SEPARATE DIGITS:
0.84147 09848 05
```

Exponent Symbol

E Specifies that the number is to be output in scientific notation. E causes the output of an E, sign of the exponent and two digit exponent. At least one digit symbol must precede the E symbol in a numeric specifier.

Example

```
10   PRINT USING "D.DDDE";125.25
20   PRINT USING "DDD.DDE";2.505
30   END
```

```
1.253E+02
250.50E-02
```

Floating Symbols

Floating symbols – S, M, X, or text in quotes – that precede all digit symbols (without a comma for separation) in a numeric specifier "float" past blanks to the leftmost digit of the number, or to the radix indicator. This is useful for output of monetary values so that the dollar sign is output next to the first digit.

Examples

```
10   IMAGE "("DDD.DD")"        ! Parentheses float
20   PRINT USING 10;2.37,10.87
30   !
40   IMAGE "$"DCDDDCDDD.DD    ! Dollar sign floats
50   PRINT USING 40;1736.73,42.35
60   !
70   !   Put comma after "$" or "(" to prevent float
80   END
```

```
  (2.37) (10.87)
   $1,736.73        $42.35
```

Sign symbols and text that are imbedded between digit symbols do not float.

Here are some examples of floating and non-floating symbols –

floating	non-floating
"$"DDD.DD	"$",DDD.DD
MDDD.DD	D"$"DD.DD
	DMDD.DD

X, S, M, or text imbedded in a numeric field stops the floating field.

Replication

Many of the symbols used to make up field specifiers can be replicated (repeated) to specify multiple symbols by placing an integer in the range 1 through 32 767 in front of the symbol. The following IMAGEs all specify the same format string –

```
10   PRINT USING "DDD.DD";123.45
20   PRINT USING "D2D.2D";123.45
30   PRINT USING "3D.DD";123.45
40   PRINT USING "3D.2D";123.45
50   END
```

```
123.45
123.45
123.45
123.45
```

Example

```
  10   OPTION BASE 1
  20   DIM A(3,3)
  30   MAT A=(2)
• 40   PRINT USING "3(D,X)/";A(*)
  50   END
```

```
2 2 2
2 2 2
2 2 2
```

Placing an integer before a symbol works exactly the same as having multiple adjacent symbols.

The following symbols **can** be replicated –

X	D
Z	*
A	/

In addition to symbol replication, an entire specifier or group of specifiers can be replicated by enclosing it in parentheses and placing an integer in the range 1 through 32 767 before the parentheses. In this manner, both K and @ can be repeated. Up to four levels of nested parentheses can be used for replication.

Example

```
10   !     Here are examples of replication
20   IMAGE 3(DD)              ! Same as 'DD,DD,DD' OR
                                '2D,2D,2D'
30   IMAGE 4(K),2(@)          ! Replicate K and @
40   IMAGE 2(3D,(2X,2D))      ! Parentheses can be nested
50   !    Rest of program follows
```

Compacted Specifier

A single symbol, K, is used to define an entire field for either numeric or string output. If the corresponding print using item is a string, the entire string is output. If it is a numeric, it is output in STANDARD form. K outputs no leading or trailing blanks.

Example

```
10   PRINT USING "K/";"AGES:"
20   IMAGE KXX,2D          ! Name can be any length
30   FOR I=1 TO 3
40      READ A$,A
50      PRINT USING 20;A$,A
60   NEXT I
70   DATA Mary,10,Hildegard,20,Amy,15
80   END
```

```
AGES:

Mary  10
Hildegard  20
Amy  15
```

Carriage Control

The CR-LF normally output when the print using list is exhausted can be altered by using a carriage control symbol as the first item in a format string; a comma must separate it from the next item.

#	Suppresses both the carriage return and linefeed.
+	Suppresses the linefeed.
–	Suppresses the carriage return.

Example

```
10   IMAGE #,4(A,2X)      ! Suppress linefeed
20   IMAGE K
30   PRINT USING 10;"A","B","C","D"
40   PRINT USING 20;"****"
50   END
```

```
A  B  C  D  ****
```

PRINT USING "+" is equivalent to PRINT LIN(0); and PRINT USING "-" is equivalent to PRINT LIN(-1);.

Field Overflow

If a numeric item requires more digits than the field specifier provides, an overflow condition occurs. When this happens, all preceding, correct items are output, followed by a CR-LF. The item which overflowed is output in STANDARD format followed by the field specifier which caused the overflow. Another CR-LF is output, then the rest of the print using list is output.

Example

```
10   PRINT USING "3(DD.DD)/";25.5,250.25,20.25
20   PRINT USING "DD.D,DDD,DD";25.5,-250.5
30   END
```

```
  25.50
• 250.25 DD.DD
  20.25

  25.5
• -250.5 DDD
```

An important thing to remember is that a minus sign not explicitly specified with S or M requires a digit position.

No error message occurs when a field overflow occurs, but the computer beeps.

Summary

Here is a summary table of image symbols and their uses –

Image Symbol	Symbol Replication Allowed?	Purpose	Comments
X	Yes	blank	Can go anywhere
" "	No	Text	Can go anywhere
D	Yes	Digit	Fill = blanks
Z	Yes	Digit	Fill = zeroes
*	Yes	Digit	Fill = asterisks
S	No	Sign	"+" or "–"
M	No	Sign	"Δ" or "–"
E	No	Exponent	Format = ESDD
.	No	Radix	Output "."
C	No	Comma	Conditional digit separator
R	No	Radix	Output ","
P	No	Decimal point	Conditional digit separator
A	Yes	Characters	Strings
( )	Yes	Replicate	For specifiers, not symbols
#	No	Carriage control	Suppress CR-LF
+	No	Carriage control	Suppress LF
–	No	Carriage control	Suppress CR
K	No	Compact	Strings or numerics
,	No	Delimiter	
/	Yes	Delimiter	Output CR-LF
@	No	Delimiter	Output FF

Considerations

One factor that must be taken into account when creating formatted output with PRINT USING is the printer width. When dealing with numeric output, format strings should be designed so that a line of characters doesn't exceed the number of characters per line of the printer. PRINT USING does not provide carriage return-linefeeds to keep lines within the width of the printer.

Advanced Printing Techniques

Advanced printing techniques on the CRT and internal thermal printer are covered in Appendix A.

Overlapped Processing

Your computer has a capability which can enable a program to run faster and more efficiently. This capability is known as overlapped processing or overlapped I/O. In overlap mode, I/O initiated by a program statement proceeds in parallel with the execution of subsequent program lines, while in serial mode the I/O is completed before the next line is executed.

Overlap mode should be used when the amount of computation time is greater than the amount of I/O time. A program that has significantly more computation time will have only a small gain from overlap mode. A program with significantly more I/O time should be run in **serial** mode. This is because extra time is needed to queue up the pending I/O operations. This time is most significant when fast peripherals are used. In overlap mode, I/O and computation statements should be intermixed.

The OVERLAP Statement

Overlap mode is set by the OVERLAP statement –

```
OVERLAP
```

If you are using ON ERROR (see Chapter 12) to trap errors, I/O errors (numbers 54-103) aren't trapped if overlap mode is in effect.

The SERIAL Statement

The computer is returned to the serial processing mode which is the default mode at power on, SCRATCH and SCRATCH A by the SERIAL statement –

```
SERIAL
```

Using serial mode is recommended during program debugging to avoid confusing results.

Accessing Color on the CRT

There are three ways to access color for printing on the CRT –

- CONTROL and a Special Function Key
- CHR$
- An escape code sequence

The primary purpose for alphanumeric color on the CRT is to increase the visual impact of output on the CRT. You can store program lines that have color in them, when the colored characters are inside a quote field or a remark.

Color Using CONTROL

The CRT special features of INVERSE VIDEO, BLINKING and UNDERLINE are accessed by pressing CONTROL and the appropriate SFK. On the facing edge of SFKs 8 thru 15 is a colored oval which represents the color which can be accessed when pressing CONTROL and the SFK. The color remains in effect until you press CONTROL and another color SFK, or until the special features are cleared, which re-instates White as the color.

Color Using CHR$

Color can be selected using a value of 136 to 143 as the CHR$ argument. The colors and their CHR$ arguments are shown in this table.

136	137	138	139	140	141	142	143
WHITE	RED	YELLOW	GREEN	CYAN	BLUE	MAGENTA	BLACK

For example,

```
PRINT CHR$(137);"A"
```

results in a red A being printed on the CRT. Once you select the color it remains in effect until you change it with another CHR$ or an escape code sequence.

Color Using the Escape Code Sequence

Color can be selected for the PRINT and PRINT USING statements only, using the escape code sequences on the System 45. The use of escape codes does not work with the DISP statement. The sequence is –

PRINT CHR$(27) & "&d specifier" followed by the items being printed.

The specifiers are –

@ – Clear (Only for items A thru G)	H – White
A – Blinking	I – Red
B – Inverse Video	J – Yellow
C – Blinking and Inverse Video	K – Green
D – Underline	L – Cyan
E – Underline and Blinking	M – Blue
F – Underline and Inverse Video	N – Magenta
G – Underline, Inverse Video and Blinking	O – Black

For example,

```
PRINT CHR$(27) & "&dKA"
```

results in a green A being printed on the CRT. Once a color is selected it remains in effect until you change it with another CHR$ or an escape code sequence.

Only one specifier is allowed in each escape code sequence, so to turn on additional highlights, you would need additional escape code sequences, as shown here.

```
PRINT "Ec&dCEc&dLBlinking Cyan Inverse Video"
```

results in the message printed in blinking, cyan inverse video. So does –

```
PRINT CHR$(131)&CHR$(140) & "Blinking Cyan Inverse Video"
```

Turning the highlights off can be done by –

```
PRINT "Ec&d@Ec&dH"
```

or by using –

```
PRINT CHR$(128)&CHR$(136)
```

Chapter 11

Mass Storage Operations

Mass storage lets you save programs and data on a mass storage medium, and retrieve them later. You use the same statements and commands to access either the internal tape cartridges or an external disc. If you want more information about mass storage than is covered in this manual, refer to the Mass Storage ROM Programming Manual.

page 172 • **MASS STORAGE IS** (specifies standard mass storage device)
page 176 • **INITIALIZE** (lets a new medium be used)
page 177 • **CAT** (lets you see what files are the medium)
page 179 • **CAT TO** (stores catalog output into a string array)
page 182 • **SAVE, GET** (stores and retrieves a program as a data file)
page 182 • **LINK** (retrieves a SAVEd program; saves variable values)
page 184 • **RE-SAVE** (stores a different version of a SAVEd program)
page 185 • **STORE, LOAD** (stores and retrieves a program as a program file)
page 186 • **RE-STORE** (stores a different version of a STOREd program)
page 187 • **CREATE** (sets up a new file for storing data)
page 188 • **ASSIGN** (opens a data file for accessing it; closes it so it can't be accessed)
page 189 • **PRINT#, READ#** (store and retrieve data into a data file)
page 194 • **MAT PRINT#, MAT READ#** (stores and retrieves arrays into a data file)
page 197 • **ON END#, OFF END#** (sets up and cancels a trap for end-of-file condition)
page 199 • **BUFFER** (reduces device access by storing data in a buffer temporarily)
page 201 • **CHECK READ, CHECK READ OFF** (sets up and cancels write verification)
page 201 • **PROTECT** (protects a file to guard against accidental erasure)
page 202 • **PURGE** (erases a file)
page 202 • **COPY** (duplicates a file)
page 203 • **RENAME** (gives a new name to a file)
page 203 • **STOREKEY, LOADKEY** (stores and retrieves Special Function Key definitions)
page 204 • **STORE BIN, LOAD BIN** (stores and retrieves binary routines)
page 204 • **STORE ALL, LOAD ALL** (stores and retrieves the entire state of the memory)
page 207 • **REWIND** (rewinds the tape cartridge)

Mass Storage Function

- TYP (determines what data type will be accessed next)

Terms

- Standard mass storage device – the device where mass storage operations are directed if not specified otherwise. It is the righthand tape drive at power on and SCRATCH A.
- File – the basic unit into which programs and data are stored. Every file has a unique name.
- Record – the smallest addressable unit on a medium.
- Directory – the medium's record of all its file information.
- Serial file access – accessing data items one after the other)
- Random file access – accessing data in a specific record in the file.

Get Started with Mass Storage

To store a program on a tape cartridge, follow these steps –

1. type `MASS STORAGE IS ":T15"`; press EXECUTE.
2. insert a cartridge into the righthand tape drive.
3. type `CAT`; press EXECUTE.
4. if you get ERROR 85, initialize the tape by typing `INITIALIZE ":T15"`; pressing EXECUTE.
5. if you get a catalog listing, note the names; you can't use a duplicate name.
6. pick a **file name**; with any 1 to 6 characters (except :, ", CHR$(0) or CHR$(255)).
7. type `SAVE` "**file name**"; press EXECUTE.

Your file is now on the tape under the file name. To retrieve it later, insert the tape into the righthand tape drive.

1. type `MASS STORAGE IS ":T15"`; press EXECUTE.
2. type `GET` "**file name**"; press EXECUTE.

Terms

The following terms are used in mass storage operations –

File number – the number assigned to a mass storage data file by an ASSIGN statement so that the file can be accessed for data storage. Its range is 1 through 10.

File name – Every file must be given a unique name. A file name is a one to six character string expression that can have any characters with the exception of a colon, quote mark, ASCII NULL (CHR$(0)), or CHR$ (255). Blanks are ignored. Here are some examples of file names –

```
"Data1"
"TEMP"
"ACCTS"
"Names"
```

Select code – The computer accesses I/O devices with a select code. It is an expression (rounded to an integer) in the range 0 through 16. The following select codes are reserved by the computer and can't be set on an interface –

- 0 Internal Thermal Printer and keyboard
- 13 Graphics option
- 14 Optional tape drive
- 15 Standard tape drive
- 16 CRT

Mass storage unit specifier (msus) – is used to direct operations to a specific mass storage device. It can be any string expression of the form –

:device type [select code [, controller address | 9885 unit code [, unit code]]]

The letters specifying the various mass storage device types are –

Letter	Device	98413A/B/C ROM	Required Interface
T	internal tape	n/a	n/a
Y	7905M (removable)	yes	98041
Z	7905M (fixed)	yes	98041
C	7906M (removable)	yes	98041
D	7906M (fixed)	yes	98041
M	7910H (fixed)	(98413B/C)	98034[1]
P	7920M (removable)	yes	98041
X	7925M (removable)	yes	98041
F	9885M/S (flexible)	yes	98032
H	9895A (flexible)	yes	98034[1]
~~Q~~	~~7908 (fixed)~~	~~(98413C only)~~	~~98034[1]~~

The **select code** can be an integer in the range 1 through 15 with 14 and 15 reserved for the tape drive and 13 reserved for graphics. If you don't specify a select code, the computer uses a default value: 15 is default for T devices, 8 for F, 7 for H devices and 12 for all others.

The **controller address** specifies a hard disc controller. It can be an integer from 0 through 7. The default controller address is 0.

The **9885 unit code** can be an integer from 0 through 3. The default unit code is 0.

The **unit code** can be an integer from 0 through 7. The default unit code is 0. It is ignored for the 9885 and tape cartridge.

Here are some examples of mass storage unit specifiers –

msus	Explanation
`":T15"`	Standard tape cartridge drive
`":T14"`	Optional tape cartridge drive
`":F8"`	9885 flexible disk at select code 8
`":Y4,0,3"`	7905A removable platter, select code 4, controller address 0, unit code 3
`":Q7,5"`	7908 disc drive, select code 7, controller address 5.

1 Only one other mass storage device may share this HP-IB interface.

Remember that the mass storage unit specifier can be any string expression. The following program segment illustrates this.

```
10    Type$="F"
20    Select_code=7
30    MASS STORAGE IS ":"&Type$&VAL$(Select_code)
40    STOP
```


File specifier – A file specifier names a file and what storage device it is on. It can be a string expression of the form – file name [mass storage unit specifier]. Here are some examples –

```
"Data:F8"
"Backup:Y4,2"
"TEMP"&":T15"
"ACCTS"&Msus$ when Msus$=":F10"
```

Protect code – A protect code can be used to protect a file from accidental erasure. It can be any valid string expression except one with a length of zero. Only the first six characters are recognized as the protect code, however.

The MASS STORAGE IS Statement

At power on and SCRATCH A, the tape cartridge drive, T15, is the **standard mass storage device** for the system. This is the device to which all mass storage operations are directed if no device is specified. The default device concept is useful in creating mass storage device-independent programs.

The standard default device is changed by executing the MASS STORAGE IS statement –

MASS STORAGE IS mass storage unit specifier

Examples

```
10    !      Examples of MASS STORAGE IS
             Don't run this program
20    MASS STORAGE IS ":T15"      ! Tape cartridge drive
30    MASS STORAGE IS ":F8"       ! Flexible disk - select code 8
40    Msus$=":Y4,0,3"
50    MASS STORAGE IS Msus$       ! 7905A removable platter -
                                    select code 4, controller
60                                ! address 0, unit code 3
70    STOP
```

Structure

All mass storage operations deal with files and records, the basic components of a storage medium.

Files

Files are the basic unit into which programs and data are stored. Storage of all files is "file-by-name" oriented; that is, all files must be assigned unique names. The form these names must take is covered in the "Terms" section at the beginning of this chapter.

There are eleven types of files –

- Program files
- Data files
- KEY files
- STOREALL files
- Binary program files
- Binary data files (Mass Storage ROM)
- Root (ROOT) files
- Backup (BKUP) files
- Data set (DSET) files
- Assembly (ASMB) files
- Option ROM (OPRM) files

Data base files

Records

Every file is composed of a varying number of records. A record is the smallest addressable unit on a mass storage medium.

There are three types of records –

1. Physical records are 256-byte, fixed units which are established when a medium is initialized. Every file starts at the beginning of a physical record; this is an important fact for minimizing wasted space on a medium when creating data files. Otherwise, you need not be concerned with physical records.
2. Defined records are established using the CREATE statement and can be specified as having any number of bytes in the range 4 through 32 767 (rounded up to an even number). A defined record is the smallest unit of storage which you can address directly.
3. A logical record, a user-level rather than machine concept, is a collection of data items which are grouped together conceptually.

When a file is established with a STORE or SAVE statement (discussed later), the computer uses as many records of 256 bytes as it needs to store the program. Logical and defined records are not used with STORE.

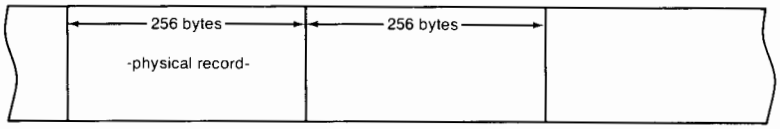
Using the CREATE statement for data files, you can specify how many defined records you wish the file to contain and how big they should be. You don't need to be concerned with the correspondence between physical and defined records, except to remember that the first defined record of a file starts at the beginning of a physical record. If a file doesn't go to the end of a physical record, space between the end of the file and the next file is wasted space.

EOF's and EOR's

Files and records are bounded on the storage medium by end-of-file (EOF) and end-of-record (EOR) marks which signify their ends. This section illustrates and describes the organization of files and records on a storage medium.

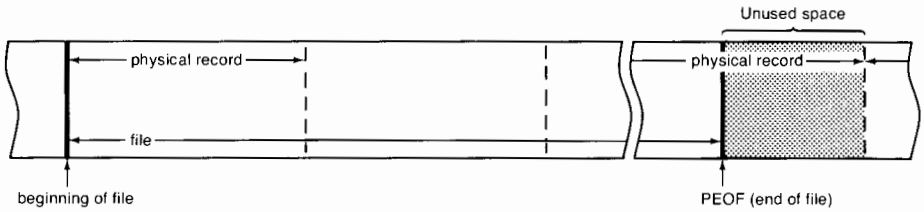
Physical Records

A storage medium is divided into 256-byte fixed physical records when it is initialized.

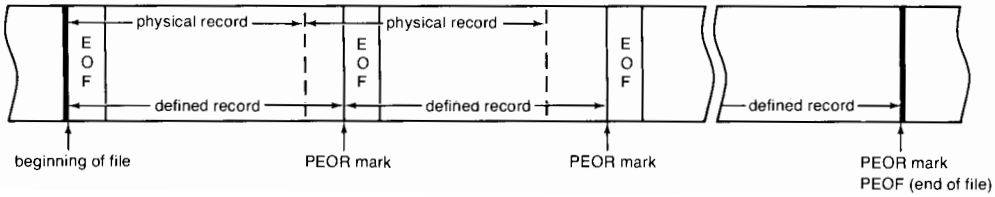


End-of-File and End-of-Record Marks

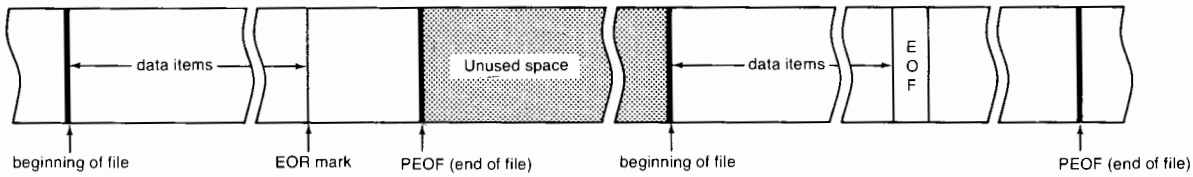
When a file is created, its end is designated by a physical-end-of-file (PEOF) mark. Any space between the PEOF and the beginning of the next physical record is unused space.



When a file is created using the CREATE statement (discussed later in this chapter), an end-of-file (EOF) mark is placed at the beginning of each defined record. Each EOF mark takes two bytes of storage space. At the same time, a physical-end-of-record (PEOR) mark is placed at the **end** of each defined record. Numeric data items can't cross a PEOR mark. If a numeric data item can't fit in the space between the previous item and the PEOR mark, it is placed in the next record, wasting the space it couldn't fit in.



As data is written to a file, the EOF marks are over-written. An EOF mark can be printed at the end of the data by printing END (see the PRINT# statement) after the data. If an EOF mark is not placed after the data, an end-of-record (EOR) mark automatically is.



Computer Museum

The Directory

The directory is the storage medium's record of all of its file information; it includes each file's name, type, length, location and loading information. The directory information is automatically revised when a file is created or purged. A spare directory is maintained on the medium in the event that the first becomes unreadable. You are warned with a message every time the spare directory is accessed if the main directory becomes unreadable. It is accessed automatically by the system when necessary. Here is the message –

```
SPARE DIRECTORY ACCESS
```

There is no provision made for recovering information stored on a medium if both directories are destroyed. If the main directory becomes inaccessible, it is wise to transfer all valuable data on the medium to another one before the spare directory is destroyed. Rewriting the main directory from the spare directory by adding, deleting or changing the name of a file may help the problem, but this is **not** a good solution. You should transfer your data.

Tape Cartridge Directory

When a tape cartridge is being used to store and retrieve information, its directory is written into memory the first time it is accessed. This is done to save wear on the tape and improve performance by accessing the directory from memory rather than from the tape. The directory on the tape is accessed only when it needs to be rewritten. The directory is erased from memory under any of the following conditions –

- Reset (CONTROL-STOP)
- SCRATCH A
- Removing the tape from the drive

The INITIALIZE Statement

The INITIALIZE statement enables a new mass storage medium to be used with Series 9800 Desktop Computers by establishing and testing physical records and main and spare directories.

A used medium can also be re-initialized; in the process, it is cleared of all information it contained previously.

INITIALIZE mass storage unit specifier [, interleave factor]

The interleave factor is a numeric expression which defines the number of revolutions per track to be made for a complete data transfer and can enable faster access to the medium. It is ignored for all devices except the HP 9885 and 9895. See the Mass Storage ROM Manual for an explanation of its use.

The INITIALIZE operation can take place at the same time as execution of a program if the program doesn't utilize the mass storage device that is involved in the initialization process. If the program attempts to use the drive on which an initialization is in progress, program execution is suspended until the operation is complete.

CAUTION

WHEN INITIALIZING TWO TAPES AT THE SAME TIME, ONE IN EACH TAPE DRIVE, YOU MUST LEAVE **BOTH** TAPES IN THE DRIVES UNTIL THE RUN LIGHT GOES OUT. DO NOT REPLACE EITHER TAPE BEING INITIALIZED WITH A TAPE WHICH HAS CURRENT DATA ON IT; THE INITIALIZATION PROCESS MAY NOT BE COMPLETE AND INFORMATION COULD BE DESTROYED.

Examples

```
10   INITIALIZE ":T15"    ! Initialize tape in standard drive
11   INITIALIZE ":T14"    ! Initialize tape in optional drive
20   A$=":F8"
30   INITIALIZE A$        ! Initialize flexible disk
40   INITIALIZE ":F7",5   ! 5 is interleave factor
50   END
```

The CAT Statement

The CAT (catalog) statement outputs a listing of directory information for a storage medium: file names, types, and physical specifications.

CAT [selective catalog specifier/msus [, heading suppression]]

CAT# select code [, HP-IB device address] [; selective catalog specifier/msus [, heading suppression]]

The selective-catalog-specifier parameter is a string expression one through six characters in length. It causes only those files whose names begin with that combination of characters to be cataloged.

The heading-suppression parameter is a numeric expression. If its value is 1, the heading of the catalog (the top two lines) is suppressed.

The second syntax directs the catalog output to the specified device.

Examples

```
10   CAT ":T15"              ! Catalog tape cartridge
20   CAT "Ab:F8"             ! Catalog all files starting with
                               'Ab' on disk at select code 8
30   CAT #6;"Dat:F8",1       ! Catalog all files starting with
                               'Dat' on disk at select code 8
40                           ! Output goes to select code 6;
                               heading is suppressed
50   MASS STORAGE IS ":F7,2"
60   CAT "S"                 ! Catalog all files starting with
                               'S' on 9885 disk at select code 7
70   END
```

The information for each file is printed on one line. Here is a sample catalog output.

```
   1      2    3      4          5          6
  NAME   PRO  TYPE  REC/FILE  BYTES/REC  ADDRESS
7 T15          8 2      3                  4

  Uri          ALL     40        256        5
  SETUP        PROG     2        256       45
  SETUP2       PROG     2        256       47
  SETUP3       PROG     2        256       49
  SEARCH       DATA     2        256       51
  STRING       DATA     2        256       53
  AIDS         KEYS     1        256       55
  MODKEY       KEYS     1        256       56
  NEWKEY       KEYS     1        256       57
  PR           DATA     2        256       58
  PRKEY        KEYS     1        256       60
```

11

1. NAME — The name given to the file when the information is stored on the medium.

2. PRO — An asterisk in this column designates a protected file.

3. TYPE — The various file types are specified by the following:

   PROG for a program file
   DATA for a data file
   KEYS for a KEY file
   ALL for a STOREALL file
   BPRG for a binary program file
   BDAT for a binary data file (Mass Storage ROM)
   OPRM reserved for additional ROM-defined file type.
   ROOT for a data base root file
   BKUP for a data base backup file
   DSET for a data base file containing a data set
   ASMB for an assembly language file

   If a medium is being cataloged that was not initialized on your particular model of Series 9800 Desktop Computer, your computer attempts to determine what types the files are and puts a question mark after the type in the catalog output for all but DATA files. The type may or may not be correct. See the Mass Storage ROM Manual for more information.

4. REC/FILE — The number of defined records in the file.

5. BYTES/REC — The number of bytes per defined record.

6. ADDRESS — The address of the physical record number with which the file begins. With the tape cartridge, it is the number of the first physical record. Knowing the length and address of files can let you find the gaps between files to see how much room is left on the medium. See the Mass Storage ROM Manual for information about other devices.

7. msus — The mass storage device on which the catalog was performed.

8. Available tracks — The number of tracks available for use. This is most important with the 9885; see the Mass Storage ROM Manual.

The CAT TO Statement

The CAT TO statement writes the specified catalog output into a one-dimensional string array. This allows your programs to have access to mass storage catalogs. CAT TO can enable you to copy files from one medium to another under program control. It can also enable you to determine what the current standard mass storage device is for the system. The CAT TO statement also causes the computer to revert to serial mode, temporarily.

CAT TO string array identifier [, skip count [, return variable]] [; selective catalog specifier/msus [, heading]]

The minimum length to dimension the array elements is 41 characters. Any elements not filled with catalog entries are filled with the null string.

The **skip-count** parameter is a numeric expression which specifies the number of catalog entries (lines) to be skipped before information is entered into the array.

The **return-variable** parameter must be a simple numeric variable. After the CAT TO operation, its value equals the number of the last catalog entry that was entered into the array. 0 is returned as its value if there are no more catalog entries at the end of the catalog that weren't entered into the array. The value of the return variable, if not 0, could be used as the next skip-count value to obtain the next part of the catalog for another CAT TO operation.

The **selective-catalog-specifier/msus** parameter is the same as for the CAT statement. The selective catalog specifier affects both the skip-count and the return-variable parameters. The skip-count skips the indicated number of entries of the selective catalog. The return-variable returns the number of the last entry of the selective catalog.

The **heading** parameter is a numeric expression which, when its value is anything other than 1, causes the second line of the standard CAT output to be entered into the first array element. Otherwise, none of the heading is entered into the array. If it is omitted, the default value is 1, so the heading is not entered.

Examples

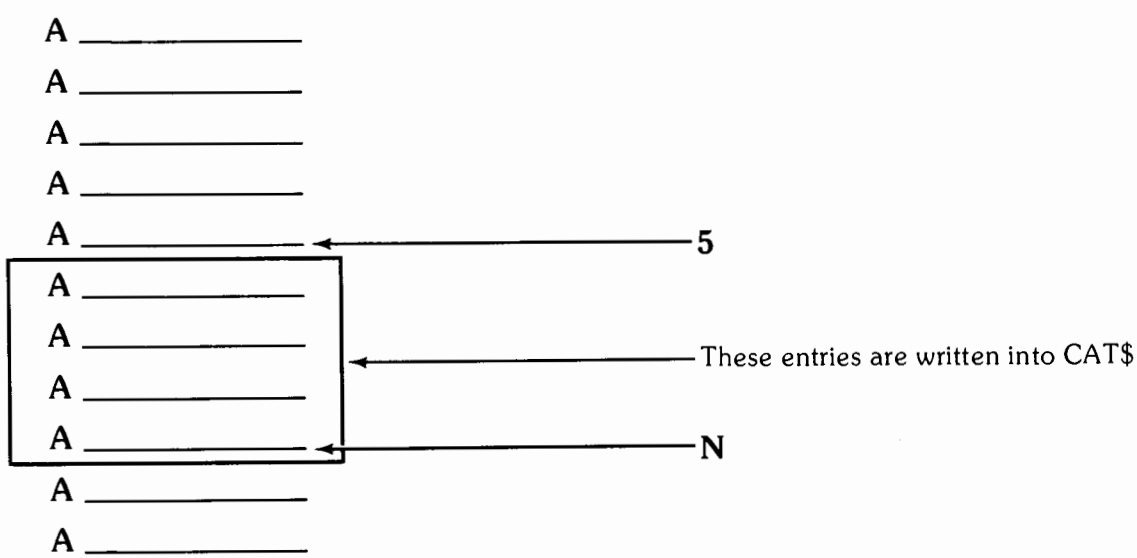
Here are some example CAT TO statements –

```
1     DIM Tape$(6)[41]
10    ! These are example CAT TO statements
20    CAT TO Tape$(*)
30    CAT TO Tape$(*),5        ! Skip 5 lines
40    CAT TO Tape$(*),5,N      ! N is return variable
50    CAT TO Tape$(*);"A",2    ! Catalog only files that start
                                with 'A'; enter the heading
60    CAT TO Tape$(*),5;"A"    ! Skip 5 entries of selective
                                catalog
70    END
```

Here is an example that illustrates how the skip-count and return-variable parameters work. Assume that the medium being cataloged has many files starting with "A" on it. This program –

```
10    OPTION BASE 1
20    DIM Cat$(4)[41]
30    CAT TO Cat$(*),5,N;"A"   ! Skip 5, N is return variable
40    PRINT N
50    END
```

creates a "window" the size of the string array around part of the catalog –



When this program is run, with an appropriate mass storage medium, 9 is returned as the value for N.

Here is an example that uses CAT TO to copy all files on a medium to another medium –

```
10      DIM A$(5)[41]
20      N=0
30 Loop:    CAT TO A$(*),N,N;":T15"
40      FOR I=0 TO 5
50          IF A$(I)<>"" THEN Doio
60          PRINT "NULL ENTRY"      ! If number of entries is
                                        not a multiple of 5
70          GOTO Regroup
80 Doio:    IF A$(I)[15;1]="?" THEN Cant_copy
90      COPY A$(I)[1;6]&":T15" TO A$(I)[1;6]&":T14"
100     PRINT A$(I)[1;6]&" COPIED FROM :T15 TO :T14"
110     GOTO Regroup
120 Cant_copy: PRINT "CAN'T COPY ";
130     PRINT A$(I)[1;6]
140 Regroup: NEXT I
150     IF N<>0 THEN Loop
160     END
```


Storing and Retrieving Programs

Programs can be stored onto a mass storage medium in two different ways, into two different types of files.

The first type of file for storing programs is known as a **data file**. When a program is stored into a data file, it is stored as a series of strings, with one string per program line. This method is not the fastest method of storing and retrieving programs, but it has a significant advantage. A program stored into a data file can be accessed as string data items by other programs. This type of file can also be used by some other HP desktop computers, such as the System 35. Programs are stored into data files with the SAVE and RE-SAVE statements and retrieved with a GET or LINK statement.

The second type of file is known as a **program file**. When a program is stored into a program file, it is stored in a compiled, internal code interpretation. Storing the program also stores all binary routines currently in memory along with the program. This is the fastest method for storing and retrieving programs. Programs are stored into program files with the STORE and RE-STORE statements and retrieved with the LOAD statement.

If you get an ERROR 2 while trying to store a large program you've just run, you can try scratching your variables with SCRATCH V to free up some memory, then storing the program.

The SAVE Statement

The SAVE statement stores the program and any subprograms in computer memory into a data file on the storage medium.

SAVE file specifier [, beginning line identifier [, ending line identifier]]

Execution of the SAVE statement creates a data file by "listing" the program and saving the list on the medium as string data, one program line per string, with a maximum length of 160 characters. In this way, the file can be read, modified, or rewritten as string data by other programs.

If you attempt to SAVE a program that has been SECUREd, the information written to the medium is meaningless.

When only the file specifier is given, the entire program is saved. If the beginning line identifier is specified, the program from that number to the end is saved. If both line identifiers are specified, the program section, from the first line identifier to the second, inclusive, is saved. If the first line identifier is a label which is in a subprogram and execution is not currently in that subprogram, ERROR 3 occurs.

Examples

```
10   SAVE "LIFE"            ! Saves program into 'LIFE' on
                              standard mass storage device
20   SAVE "LIFE:F8",40      ! Saves program starting at line 40
                              into 'LIFE' ,on flexible disk
30   A$="LIFE"
40   SAVE A$,100,300        ! Saves lines 100-300 of program
                              into 'LIFE'
50   STOP
```

The GET Statement

The partner of the SAVE statement, the GET statement retrieves and puts into memory a program saved previously with the SAVE statement, or any string data file consisting of valid BASIC statements preceded by line numbers, stored one line per string.

GET file specifier [, line identifier [, execution line identifier]]

Execution of the GET statement causes the computer to read the specified data file and expect to find a succession of strings that are valid program lines. As the program is retrieved, each line is read in and syntax checked to make sure it is a valid line. If GET was executed from a program, any tracing which was in effect is cancelled.

If no line identifiers are specified, the entire stored program is loaded into computer memory, destroying any programs or data (except data stored with COM) in memory.

If one line identifier is specified, the program is renumbered as it is loaded so that it begins with the number of the specified line of the program currently in memory. Any lower-numbered lines from a previous program are retained. The numbering remains the same on the storage medium.

If the GET was executed in a program, program execution is restarted with –

- The program line immediately following the GET statement in the original program or with
- the first line of the loaded program if there were no lines after the GET statement or if these lines were destroyed by the GET statement.

If two line identifiers are specified, program execution is restarted with the second line identifier.

When a program retrieved with GET has an invalid line in it, the invalid line and an error message are listed on the standard system printer. An example of how this can occur is when a program is SAVEd with the Mass Storage ROM installed in the machine and later retrieved with GET when the ROM is not installed. Any lines which have mass storage unit specifiers other than :T15 or :T14 are listed with an error message.

Examples

```
10   GET "SMALL"       ! Program in 'SMALL' is retrieved
20   GET "SMALL",75    ! Program in 'SMALL' is retrieved and
                         renumbered to begin with 75
30   A$="SMALL:F8"
40   GET A$,100,10     ! Program in 'SMALL' is retrieved and
                         renumbered to begin with 100
50                     ! Execution begins with line 10 in memory
60   STOP
```

The LINK Statement

The LINK statement is identical to the GET statement discussed previously, except that the current values of all variables are retained.

LINK file specifier [, line identifier [, execution line identifier]]

If no line identifiers are specified, the program is loaded, destroying the current program in memory.

The first line identifier specifies that the loaded program is to be renumbered and is to begin with the line number of the specified line.

If two line identifiers are specified, execution begins with the second line specified.

In effect, GET performs a RUN operation on the loaded program, whereas LINK performs a CONT operation, involving no pre-run initialization of variables.

Examples

```
10   LINK "LIFE"          ! Program in 'LIFE' is retrieved
20   LINK "LIFE",50       ! Program in 'LIFE' is retrieved
                            and renumbered to begin with 50
30   A$="LIFE:F8"
40   LINK A$,100,10       ! Program in 'LIFE' is retrieved
                            and renumbered to begin with 100
50                        ! Execution begins at line 10 in memory
60   STOP
```

The RE-SAVE Statement

A program stored in a data file can be loaded into memory and edited. It can then be re-saved into the same file using the RE-SAVE statement –

RE-SAVE file specifier [, protect code] [, beginning line identifier [, ending line identifier]]

RE-SAVE is equivalent to PURGE followed by SAVE.

The protect code is used only if the file has been protected. When no line identifiers are specified, the entire program is saved. When one line identifier is specified, the program is saved from that line to the end. When two line identifiers are specified, that block of lines is saved.

NOTE

If you are attempting to RE-SAVE a program you've made longer and get ERROR 64, that means there is no space to save the new version. If you were writing to a tape cartridge, the old version is still there. However, if you were writing to any type of disk, the old version is erased.

Computer Museum

Examples

```
10   RE-SAVE "LIFE"          ! Purge program in 'LIFE' & save
                               program in memory into it
20   RE-SAVE "LIFE",50       ! Purge program in 'LIFE' & save
                               current program; start at line 50
30   A$="LIFE"
40   RE-SAVE A$,100,250      ! Purge program in 'LIFE' & save
                               lines 100-250 of current program
50   RE-SAVE "LIFE","SM"     ! Resave 'LIFE' with protect
                               code 'SM'
60   STOP
```

The STORE Statement

The STORE statement creates a program file and stores the program and any binary routines in memory into it.

STORE file specifier

Examples

```
10   STORE "WAGE"          ! Stores program into 'WAGE' on
                             standard mass storage device
20   STORE "WAGE:F8"       ! Stores program into 'WAGE' on
                             flexible disk at select code 8
30   A$="WAGE"
40   STORE A$              ! Stores program into 'WAGE'
50   STOP
```

The LOAD Statement

Programs saved with STORE are retrieved with the LOAD statement.

LOAD file specifier [, execution line identifier]

Execution of the LOAD statement destroys any program, binaries and data in memory and loads the program and any binary routines. However, any data stored in common is preserved if the loaded program has a COM statement. If the LOAD statement comes from the keyboard and no line identifier is specified, control returns to the keyboard after loading. If it comes from execution of a program line in memory, execution begins at the first line of the loaded program.

NOTE

A program which includes enhanced or color graphics keywords and has been STOREd on a 9845B Model 2XX or a 9845C cannot be LOADed on a 9845B Model 1XX.

When the line identifier is specified, execution of the loaded program begins at that line.

Examples

```
10   LOAD "Chek"          ! Loads program in 'Chek' into memory
20   LOAD "Chek",50       ! Loads program in 'Chek' into memory;
                            execution begins with line 50
30   A$="Chek:F8"
40   LOAD A$              ! Loads program in 'Chek' on flexible
                            disk at select code 8 into memory
50   STOP
```

When LOADing a program which includes statements that are enabled by more than one ROM (for example – TDISP, enabled by the I/O and Datacomm ROMs), be certain that your computer has the same ROMs as the one used when the program was STOREd.

The RE-STORE Statement

A program file can be loaded into memory and edited, then re-stored into the same file using the RE-STORE statement –

RE-STORE file specifier [, protect code]

RE-STORE is equivalent to PURGE followed by STORE.

The protect code is used only if the file has been protected.

Examples

```
10   RE-STORE "PAYROL"  ! Purge program in 'PAYROL' and store
                          program in memory into it
20   A$="PAYROL:F8"
30   RE-STORE A$,"XX"   ! Store program into 'PAYROL' with the
                          protect code 'XX'
40   STOP
```

Storing and Retrieving Data

Data in the form of numbers and strings can be stored into a data file. This is the same type of file as the one created by the SAVE statement, but it is created differently. You can group conceptually related data items together, forming what is known as a logical record.

There are five basic data file operations –

- Creating a file – CREATE
- Opening a file – ASSIGN
- Recording data – PRINT# and MAT PRINT#, random and serial
- Retrieving data – READ# and MAT READ#, random and serial
- Closing a file – ASSIGN

Considerations

Before data is written onto a storage medium it is held in a buffer in the computer, then written to the medium when the buffer is full. You should take care to terminate your program properly before removing your tape or disk from the drive; otherwise data may still be in the buffer, not written to the medium. You can make sure this doesn't happen by closing the file, pressing (STOP) or reaching a STOP or END in your program.

The CREATE Statement

The CREATE statement is used to create a data file.

CREATE file specifier, number of defined records [, record length]

The number-of-defined-records parameter is a numeric expression in the range 1 through 32 767.

The record-length parameter is a numeric expression, specifies the length of a defined record in bytes and is rounded up to an even integer. Its range is 4 through 32 767. If it is not specified, a defined record length of 256 bytes is assumed.

The size of a file created is limited by the amount of available space on the medium. A medium overflow error (ERROR 64) occurs if more records are specified than the medium can hold.

CREATE also puts an EOF mark in the first two bytes of every defined record.

Examples

```
10   !    These are example CREATE statements
20   CREATE "DATA:F8",10    ! Create a file named 'DATA';
                              10 records of 256 bytes
30   CREATE "Names",15,50   ! Create a file called 'Names';
                              15 records of 50 bytes each
40   STOP
```


When creating data files, you must be sure that the length and number of your defined records suit the storage requirements of the logical records you plan to store. If the next data item in a list being recorded won't fit in the space between the previous item and the end of the record, that space will be wasted. To determine storage requirements, see the section on Data Storage which is later in this chapter. Attempts to store data into an insufficient amount of storage space results in an error.

The ASSIGN Statement

Data files must be opened before they can be accessed. This is done with the ASSIGN statement. The two syntax shown below are equivalent.

ASSIGN file specifier TO# file number [, return variable [, protect code]]
ASSIGN# file number TO file specifier [, return variable [, protect code]]

The ASSIGN statement sets up or references an existing internal **files table** and allows you to utilize data files (with PRINT# and READ# statements). The files table has room for ten entries. All entries are cleared when a program is run, and when SCRATCH, SCRATCH V, SCRATCH C, SCRATCH A or reset is executed. The file number is a numeric expression; its range is 1 through 10. The ASSIGN statement also assigns a **file pointer** used for data access to the file number, and positions the pointer at the beginning of the first record of the file.

The optional return variable can be a simple numeric variable or array element and is set after execution to indicate various results. You can use its value to check for errors. If no return variable is specified, an error occurs if the file isn't found, is protected or is of the wrong type.

Return Variable	Meaning
0	File available
1	No such file found
2	File is protected, or wrong file type

The protect code is a string expression, and is necessary only if the file was protected earlier. For all disks it must be the same protect code as the one to protect the file. If the file isn't protected, including the protect code causes an error. Using the null string as a protect code corresponds to an unprotected file.

Examples

```
10   ASSIGN #1 TO "Data"
20   ASSIGN "SCORES" TO #4,Return
30   ASSIGN "SCORES" TO #5
```


Line 20 illustrates a return variable. Lines 20 and 30 show that more than one number can be assigned to a file.

All file numbers must be assigned prior to referencing them with PRINT #, TYP, and READ #. This includes subroutines and function calls which have as their parameters fill numbers.

Serial File Access

Serial file access is used to store or retrieve data items one after the other, without regard to defined records. Logical records can be longer or shorter than defined record length. For each data file opened, a file pointer keeps track of the data item currently being accessed. As you store or retrieve data, the pointer moves serially forward through the file.

The PRINT# Statement – Serial

The serial PRINT# statement records values onto the specified file from the specified variables or strings in computer memory.

PRINT# file number; data list [, END]
PRINT# file number; END

The data list is a collection of items separated by commas. The items can be variables, array identifiers, numbers, or strings of characters. The last or only item can be END, which causes an EOF mark to be printed. Otherwise, an EOR mark is placed after the data list is printed.

Printing begins at the position of the pointer after the data item most recently stored or retrieved, or at the beginning of the file if nothing has been stored or retrieved, or if the pointer has been repositioned to the beginning of the file.

When storing a long string, it might be too long to be contained in one defined record. In that case, the string is automatically broken up and stored into as many defined records as it needs. This adds four bytes to the amount needed to store the string each time the string crosses over into another defined record. The parts of the string are identified as first, intermediate, or last.

The length of data in the list must equal or be less than the storage space that remains in the file after the pointer; otherwise, an EOF error occurs, signaling that you have filled your file. Data can also be stored using the PRINT# statement in a file created with the SAVE statement if the file has been assigned a number. SAVE, in effect, performs a serial print onto a file.

Examples

```
  10   CREATE "TEMP",4        ! A 4-record file
  20   ASSIGN #3 TO "TEMP"    ! #3 is assigned to it
  30   A=B=C=D=RND
  35   DIM E(2,2)
  40   MAT E=(5)
• 50   PRINT #3;A,B,C         ! Record 3 values
• 60   PRINT #3;D,E(*)        ! Record 10 values
  70   END
```

Lines 50 and 60 record values for A, B, C, D, and E (*) onto file #3. This data constitutes a written record. The EOR which was placed after the data when line 50 was executed is overwritten when line 60 is executed. Another EOR is printed after the data in line 60. Remember, an EOR signifies that there is no more data between the file pointer and the end of the defined record.

The serial PRINT# statement can also be used to generate program lines into a file. Such a file can be retrieved with GET. Here's an example –

```
  10   CREATE "PRGRM",3,50
  20   ASSIGN #1 TO "PRGRM"
• 30   PRINT #1;"10   X=5","20   Y=7"
• 40   PRINT #1;"30   PRINT X,Y,X*Y","40   END"
  50   GET "PRGRM"
  60   END
```

When this program is run, the output is –

```
 5                    7                    35
```

Executing LIST produces –

```
  10   X=5
  20   Y=7
  30   PRINT X,Y,X*Y
  40   END
```

The READ# Statement – Serial

The serial READ# statement retrieves values for variables and strings of characters from the specified file. In serial mode, EOR marks are ignored and the file pointer skips to the next record to find data.

READ# file number; variable list

Before you can re-use data which has been stored in a data file with a PRINT# statement, you must read the data back into computer memory. The data is not erased from the file; it is merely copied into the variables specified in the same order in which it was stored with the PRINT# statement. Therefore, variables do not have to have the same names specified in the PRINT# statement, but they must be of the same type. Reading begins after the last item printed or read on the specified file. To begin reading from the beginning of the file, you must reposition the pointer or do another ASSIGN.

In order to retrieve all of the information stored, your READ# statement(s) data list must match in number and type (string vs. numeric) the PRINT# statement(s) data list previously stored. If the READ# statement list specifies more data items than were originally stored, an EOR (or EOF if END was printed) error occurs, meaning there is no more data.

Data that is read must correspond to the type – numeric or string – that was printed. Precision (for numeric data items) is automatically converted for short to real, but not for real to short. You can also print an array and read back simple variables or other arrays, and vice versa.

Data in the form of strings can be read from a file created with the SAVE statement. This can be done by reading a series of as many strings as there are program lines. Each string must be long enough to contain the program line. Dimensioning the strings to 160 characters ensures this.

Examples

```
  10   F=2.17598824
  20   C=3
  30   PRINT "FULL-PRECISION:",LIN(1),"F=";F,"C=";C
  40   CREATE "XX",4          ! Create a 4-record file
  50   ASSIGN #1 TO "XX"
• 60   PRINT #1;F,C           ! Record values of F & C into file
  70   ASSIGN #2 TO "XX"
  80   SHORT B,D
• 90   READ #2;B,D            ! Read values for short-precision
                                variables
  100                         ! Values are read from full-
                                precision values
  110  PRINT "SHORT-PRECISION:",LIN(1),"B=";B,"D=";D
  120  END
```

```
FULL-PRECISION:
F= 2.17598824          C= 3
SHORT-PRECISION:
B= 2.17599             D= 3
```

Notice that value of F is rounded when used as the value for B.

However, an overflow or underflow can occur. This is illustrated by the following program.

```
  5    ASSIGN #1 TO "XX"        ! File created in last example
• 15   PRINT #1;2.5E99          ! Prints full-precision number
  25   SHORT A
  35   ASSIGN #2 TO "XX"
• 40   READ #2;A                ! Reads short-precision number
  55   PRINT A
  65   END
```

This causes `ERROR 21 IN LINE 40` to be displayed and a beep to occur. To avoid the error, `DEFAULT ON` can be executed. Then the default value is used.

Here's an example of corresponding serial `PRINT#` and `READ#` operations -

```
  10   FIXED 2
  20   OPTION BASE 1
  30   DIM A(24)
  40   MAT A=(PI)                 ! Assign values to A
  50   ASSIGN #1 TO "XX"          ! This file created in
                                    earlier example
• 60   PRINT #1;X,Y,A(*)          ! Print 26 values
  70   PRINT "X,Y & A(*):";X,Y,A(*),
  80   DIM C(12),D(12)
  90   ASSIGN #6 TO "XX"
• 100  READ #6;C(*),D(*),E,F      ! Read 26 values
  110  PRINT "C(*),D(*),E & F:";C(*),D(*),E,F
  120  END
```

Notice that 26 items are printed and 26 are read; they don't need to match as far as simple or array variable goes. Arrays are stored as a series of single data items with no regard to dimensionality.

Random File Access

Random file access is used to store or retrieve data items from a specific defined record.

Random file access requires you to specify with a numeric expression, the defined record you wish to access. The pointer is positioned at the beginning of that defined record.

The PRINT# Statement – Random

The random PRINT# statement is like the serial PRINT# statement except that it records data onto the file starting at the beginning of the specified record. However, EOF marks at the end of records aren't ignored. The data can't be larger than the record.

PRINT# file number, defined-record number[;data list [, END]]
PRINT# file number, defined-record number [; END]

The data list is identical to that used in the serial PRINT# statement. The random PRINT# statement records data into the specified record of the file. Printing starts at the beginning of the specified defined record. Any previous data in the record is overwritten. Any data not overwritten because the new logical record is shorter is inaccessible via that pointer. Specifying END causes an EOF mark to be printed after the data or at the beginning of the record (second syntax).

The written record set down by the list(s) of data must fit in the defined record, otherwise an EOR error occurs. If you attempt to specify a defined record number greater than the number specified in the CREATE statement, an EOF error occurs.

When no data list or END is specified, an EOR is printed in the first two bytes of the record, which makes the data in that record inaccessible.

Example

```
  10   ASSIGN #1 TO "XX"        ! This file created earlier
  20   A=B=C=D=RND
● 30   PRINT #1,1;A,B-2         ! Print 2 values in record 1
● 40   PRINT #1,2;C/2,D*3       ! Print 2 values in record 2
● 50   PRINT #1,3               ! Print EOR in 1st word of
                                  record 3
  60   END
```

Records 1 and 2 each have two values in them. Record 3 has an EOR in the first word.

The READ# Statement – Random

The random READ# statement is like the serial READ# statement except that reading of data into the computer begins at the beginning of the specified defined-record and won't read past an EOR or EOF mark.

READ# file number, defined record number [; variable list]

Again, as in the serial READ# statement, the variables into which you read values do not necessarily have to have the same names or precision type specified in the PRINT# statement.

If the number of items making up the data list is greater than the data in the defined record, an EOR error occurs.

Example

```
  10   !      Examples of random READ#
  20   ASSIGN #1 TO "XX"        ! File created earlier
● 30   READ #1,1;A,B            ! Read 2 values from
                                  record 1
● 40   READ #1,2;X,Y            ! Read 2 values from
                                  record 2
  50   PRINT A,B,X,Y
  60   END
```

These two operations retrieve the data stored in the previous example.

Repositioning the Pointer

If the data list is omitted, the pointer is repositioned to the beginning of the specified record. To reposition the pointer to the beginning of a file (for use with serial file access) execute –

READ# file number, 1

The MAT PRINT# and MAT READ# Statements

Entire arrays can be stored and retrieved, using either serial or random access, by use of the MAT PRINT# and MAT READ# statements.

MAT PRINT# file number [, defined record number]; array variable
[, array variable...] [, END]

MAT READ# file number [, defined record number]; array variable
[(redim subscripts)] [, array variable [(redim subscripts)] , ...]

Arrays are stored and retrieved element by element without regard to dimensionality with the last subscript varying fastest.

Example

```
10   ASSIGN #1 TO "XX"     ! File created earlier
20   OPTION BASE 1
30   DIM A(2),B(4),X(2),Y(2),Z(2)
40   MAT A=(121)
50   MAT B=(212)
60   MAT PRINT #1,3;A,B    ! Print six values
70   MAT PRINT A;B;
80   MAT READ #1,3;X,Y,Z   ! Read six values
90   MAT PRINT X;Y;Z;
100  END
```

```
 121  121

 212  212  212  212

 121  121

 212  212

 212  212
```

Arrays can also be printed and read with the PRINT# and READ# statements. Lines 60 and 80 above could also read –

```
60   PRINT #1,3;A(*),B(*)
80   READ #1,3;X(*),Y(*),Z(*)
```

Random vs. Serial Method

The decision to choose random or serial methods depends upon the structure of the data which is to be recorded and retrieved. Serial file usage makes the most efficient use of the storage medium by packing all data tightly in the file. However, the data must be retrieved from the beginning of the file and therefore an item in the middle of a file cannot be accessed until all data coming before it is accessed. Random file usage is less efficient in its use of the storage medium but it provides access to data at various points (logical records) within the file without previously accessing the data which comes before.

Closing a File

The ASSIGN statement is also used to close a file; any subsequent attempts to access that file number results in an error. It is recommended that a file **always** be closed when you are done accessing it during a program segment. The two syntax shown below are equivalent.

```
ASSIGN * TO # file number
ASSIGN# file number TO *
```

11

Other Data File Operations

The other operations which can be performed with data files are –

- determining data type – TYP function
- trapping EOR and EOF conditions – ON END#
- determining data storage requirements
- buffering a file – BUFFER#

The TYP Function

The type function is used to determine what type of data the pointer will access next.

```
TYP (-] file number)
```

The possible values for the function and their meanings are –

Value	Meaning
0	Option ROM missing or data pointer lost.
1	Full-precision number.
2	Total string
3	End-of-file mark
4	End-of-record mark
5	Integer-precision number
6	Short-precision number (If short precision was not expected, 6 indicates lost data pointer)
7	Unused
8	First part of a string
9	Middle part of a string
10	Last part of a string

If the file number is negative, the data pointer doesn't move. If it is positive, the pointer moves forward until it is positioned at something other than an EOR mark. In effect, a negative file number causes a random read. A positive file number causes a serial read, ignoring EOR marks.

The ON END# Statement

Normally, encountering an EOF or EOR during a random access READ# or PRINT# operation or encountering an EOF during serial access causes the program to stop. The ON END# statement is a declarative[1] which causes a branching operation to occur when an EOF or EOR is encountered.

ON END# file number GOTO line identifier
ON END# file number GOSUB line identifier
ON END# file number CALL subprogram name

Specifying ON END disables OVERLAP mode for that file. The routine branched to should service the EOF or EOR condition.

Example

```
  10   DIM A(5,5),B(5,5),Q(100)
  20   MAT A=(5)
  30   MAT B=(8)
  40   CREATE "DATA1",4
  50   CREATE "DATA2",4
  60   ASSIGN #3 TO "DATA1"
  70   ASSIGN #4 TO "DATA2"
  80   PRINT #3;A(*)        ! Print 36 values
  90   PRINT #4;B(*)        ! Print 36 values
  100  ASSIGN #1 TO "DATA1"
• 110  ON END #1 GOTO Reposition
  120  FOR I=1 TO 44
  130     READ #1;Q(I)
  140     PRINT Q(I);
  150  NEXT I
  160  STOP
• 170 Reposition:  ASSIGN #1 TO "DATA2"
  180  !    This subroutine opens another data file to
            access more values when I=37
  190  GOTO 130
  200  END
```

When an EOF is encountered while reading values, another file, "DATA2" is opened and used.

ON END is suspended during an INPUT, LINPUT or EDIT response request. ON END can always interrupt ON KBD, ON INT and ON KEY routines. ON END can interrupt an ON ERROR routine if they are declared in the same program segment.

1 More information about interrupt branching can be found in Chapter 14.

The OFF END# Statement

An ON END# declarative is deactivated with the OFF END# statement. OFF END also reactivates OVERLAP mode for the file if it had been in effect previously.

OFF END# file number

EOR Errors

To recover from EOR errors, you can either shorten the data in precision or amount, or purge and recreate the file with the defined records longer or more numerous.

Example

The following example illustrates a condition in which an EOR condition is generated.

```
10   CREATE "SHUN",2,16
20   ASSIGN #1 TO "SHUN"
30   DIM A$[20]
40   A$="ABCDEFGHIJKLMNOPQR"
50   PRINT #1,1;A$        ! A$ is too long for record
60   END
```

Execution causes an EOR condition (ERROR 60); A$ is longer than the record. The EOR condition can be avoided by increasing the number of bytes in "SHUN" or changing line 50 to read PRINT# 1;A$.

The following example shows how an EOF can be generated.

```
10 ! CREATE "IVNESS",5,10
20   ASSIGN #2 TO "IVNESS"
30   DIM B$(6)[2]
40   FOR I=1 TO 6
50      B$(I)=CHR$(I+32)
60      PRINT B$(I)
70      PRINT #2;B$(I)    ! Trying to print too much data
80   NEXT I
90   END
```

An EOF is generated when I = 6, B$(6) is "after" the end of file IVNESS.

Data Storage

When storing data, it is possible to optimize the use of your storage medium by minimizing the amount of unused space. The best way to do this is to create your files so they are suited to the amount of data you wish to store and to storage medium capacities.

The following tables indicate how many bytes are needed to store various variables on a mass storage medium.

Single Variable	
Full precision	8 bytes
Short precision	4 bytes
Integer precision	4 bytes
String	1 byte per character + 4 bytes + 4 bytes each time string crosses into a new defined record. (256)

Array Variable	
Full precision	8 bytes × dimensioned number of elements
Short precision	4 bytes × dimensioned number of elements
Integer precision	4 bytes × dimensioned number of elements
String	4 bytes per element + total needed for all strings as defined above.

Computer Museum

By summing up how many bytes of storage your data requires, you can tailor your file and defined record lengths to suit your needs and minimize waste. However, keep in mind that a file always begins on a new physical record. If a file requires a total of 520 bytes (2 physical records plus 8 bytes), 248 bytes are unused, and therefore, are wasted space.

The BUFFER Statement

The BUFFER statement is used to attach a buffer from user Read/Write Memory to a file number to reduce device wear and increase efficiency by reducing device transfers.

BUFFER# file number

The BUFFER statement allocates buffers from the main user Read/Write Memory by attaching a 256-byte, semi-permanent buffer to the specified file number. PRINT# statements cause transfers to the buffer (rather than to the actual medium); when the buffer is full, its contents are dumped to the medium. READ# statements fetch data from the buffer until it is exhausted; the buffer is then refilled from the medium.

Buffering files is most advantageous if all files being accessed on a specific device are buffered. See the Mass Storage ROM Manual for more information on buffering and its implications.

A buffer that is assigned to a file number is also dumped under these conditions –

- ASSIGNing that number to a different file
- A PAUSE, (PAUSE), STOP or END

All buffers are dumped when any ASSIGN is done.

A buffer is returned to main Read/Write Memory under these conditions –

- RUN
- SCRATCH A
- END
- Reset
- STOP
- Closing the file
- Returning from the subprogram in which the file being buffered was opened.

The BUFFER statement can't be executed from the keyboard.

The CHECK READ Statement

The CHECK READ statement is used to verify information written to a storage medium.

CHECK READ [# file number]

When no file number is specified, all storage operations are verified. The file number causes only PRINT# operations to that file to be verified. This is a bit-for-bit comparision.

CHECK READ has the additional function of forcing transfer to the medium of the current data record after every PRINT# operation. However, the BUFFER statement has precedence over CHECK READ. The data record is verified only when the buffer allocated by the BUFFER statement is dumped to the actual medium.

The CHECK READ operation reduces the speed of operations and increases wear on the tape cartridge. Use only when necessary.

The CHECK READ OFF Statement

The CHECK READ operation can be cancelled by executing the CHECK READ OFF statement.

CHECK READ OFF [# file number]

The PROTECT Statement

The PROTECT statement is used to guard a file against accidental erasure, especially with disks.

PROTECT file specifier, protect code

The file specifier must specify an established file on a device.

The protect code is any valid string expression except the null string. Only the first six characters are recognized as the protect code.

Examples

```
10   !    These are examples of PROTECT statement
20   PROTECT "DATA",Date$
30   PROTECT "NAMES:F8","XXX"
40   END
```

NOTE

For tape cartridges, the directory doesn't retain the protect code itself, but only notes the fact that you have protected the file. For all other mass storage devices, the protect code itself is kept in the directory. A file on the tape cartridge can be purged using any protect code; it need not be the one it was protected with.

The PURGE Statement

The PURGE statement eradicates any file (program, data, etc.) by removing its name from the name table in the directory, thereby preventing any access to the file.

PURGE file specifier [, protect code]

The protect code is necessary only if the file was previously protected. The records of the file are then returned to "available space", being combined with adjacent available records, if any.

Examples

```
10   !      These are examples of PURGE statement
20   PURGE "TEMP"
30   PURGE "EXTRA:F8"
40   PURGE "BACKUP","KEY"        ! File was protected
                                     at some other time
50   END
```

The COPY Statement

The COPY statement is used to copy the information in a file into another file.

COPY source file specifier TO destination file specifier [, protect code]

The protect code is necessary only if the source file is protected.

Execution of the COPY statement causes all records of a file to be copied. The first file specified can be of any type. A check of the name of the destination file is made; an error is given if the name is present. If not, a file of the same characteristics as the source file is created. The same storage medium can be both source and destination. If an option ROM file is copied to or from a tape cartridge, its type is changed to 'OPRM'.

Examples

```
10   !     These are examples of COPY statement
20   COPY "FILE" TO "BACKUP:F8"
30   COPY "DATA1" TO "DATA2"          ! Can be same medium
40   COPY "PROGA" TO "PROGB","***"    ! 'PROGA' was protected
                                        at some other time
50   END
```

The COPY statement is very useful for duplicating a storage medium. Each file can be copied individually, thus duplicating the entire medium.

The RENAME Statement

The RENAME statement is used to give a file a different name.

Computer Museum

RENAME old file specifier TO new file name [, protect code]

Examples

```
10   !    These are examples of RENAME statement
20   RENAME "JUNE1" TO "JUNE 2"
30   RENAME "TEMP" TO "FINAL","TRIAL" ! 'TEMP' was protected
40   END
```


STORE KEY and LOAD KEY

The typing-aid definitions of all special function keys can be stored onto a mass storage medium using the STORE KEY statement.

STORE KEY file specifier

This creates a "KEY" file.

The stored definitions can be loaded back into the keys by executing the LOAD KEY statement –

LOAD KEY file specifier

Examples

```
10   ! These are examples of STOREKEY and LOADKEY
20   STORE KEY "TEMPKY"
30   STORE KEY "AIDS:F8"
40   LOAD KEY "TEMPKY"
50   LOAD KEY "AIDS:F8"
60   END
```

STORE BIN and LOAD BIN

All binary routines currently in memory can be recorded separately from programs using the STORE BIN statement.

STORE BIN file specifier

Stored binary routines are retrieved and added to current binary routines using the LOAD BIN statement.

LOAD BIN file specifier

Examples

```
10   !  These are examples of STOREBIN and LOADBIN
20   STORE BIN "ROUTIN"
30   STORE BIN "COMPIL:T15"
40   LOAD BIN "ROUTIN"
50   LOAD BIN "COMPIL:T15"
60   END
```

STORE ALL and LOAD ALL

The entire user Read/Write Memory state: programs, variables, keys, binaries, CRT display – can be stored into a special memory file. The files table is not stored into the STORE ALL file, however.

STORE ALL file specifier

The file created by the STORE ALL statement is very large; the minimum is 38 records. STORE ALL can't be executed during execution of a subprogram.

Memory can be returned to the state it was in previously by using the LOAD ALL statement.

LOAD ALL file specifier

All files being used when the corresponding STORE ALL was executed must be reassigned.

NOTE

In order to LOAD ALL a STORE ALL file, your computer must be identical (options and memory size) to the one used when the STORE ALL was executed.

Examples

```
REM    These are examples of STOREALL and LOADALL
STORE ALL "MEMORY"
STORE ALL "2/3/78:F8"
LOAD ALL "MEMORY"
LOAD ALL "2/3/78:F8"
END
```

The Tape Cartridge

This section covers general information for using the tape cartridge for mass storage operations.

For heavy usage of mass storage files, such as nonconsecutive file sorts or data base management applications, flexible disks or hard discs are recommended for optimum performance and reliability.

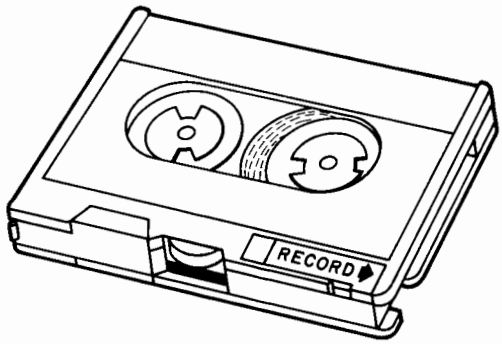
The standard tape drive is on the right hand side of the computer and is the default mass storage device at power on and SCRATCH A. Its mass storage unit specifier is :T15. The optional tape drive is on the left hand side of the computer. Its mass storage unit specifier is :T14.

NOTE

Occasionally when using the tape cartridge, unexpected high-speed movements may occur. Ignore these; they in no way affect usage, but merely assure proper tape tension.

Recording on the Tape

To record on the tape cartridge, the record tab must be in the rightmost position, in the direction of the arrow (as shown).

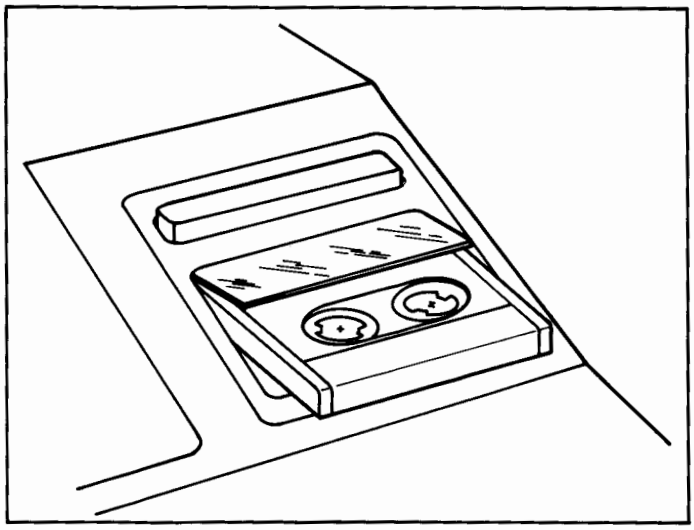


Write Protection

If the record tab is moved to the left, no information can be written to the tape. Information can only be read from the tape.

Inserting and Removing the Tape Cartridge

Insert the tape cartridge so that its label is up and the open edge is toward the computer. Both the drive window and the door beneath it open when the cartridge presses on the lower door; the cartridge can then be inserted.



To remove the tape cartridge, press the eject bar. If it is pulled out without pressing the eject bar, another cartridge can't be inserted until the eject bar is pressed.

General Tape Cartridge Information

Mass storage unit specifier	:T15 (standard tape drive)
	:T14 (optional tape drive)
Tape capacity	847 user-accessible physical records (216 832 bytes)
	42 files (directory entries)
Rewind time	19 seconds
Initialization time	3 minutes
Tape length	42.67m (140 feet)
Number of tracks	2 independent tracks
Access rate (search speed)	11 770 bytes/second
Transfer rate	1 438 bytes/second
Typical tape life	50-100 hours
Typical error rate[1]	< 1 in 10^7 bytes

1 This is dependent on the cleanliness of the tape head, tape care, and the cleanliness of the environment.

The REWIND Statement

The REWIND statement rewinds the tape to its beginning.

REWIND [mass storage unit specifier]

If no parameter is specified, the default device is used. If it is not a tape cartridge, the statement is ignored. There is also a Special Function Key to rewind each of the tape cartridges.

Operations which do not involve the tape cartridge can take place while the tape rewinds.

Mass Storage Errors

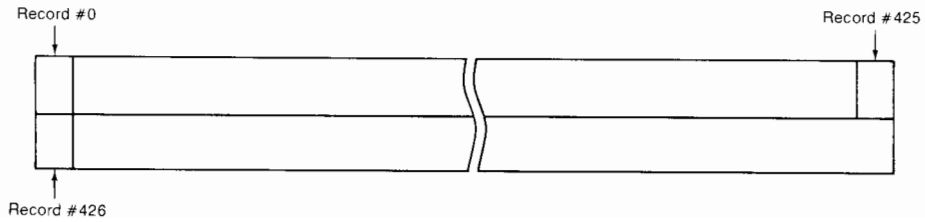
When using the tape cartridge, wear caused by contact between the tape and the read/write head can occur. If at any time, the tape makes rattling sounds while moving, or error 84, 87, 88 or 89 or a SPARE DIRECTORY ACCESS warning begin to occur frequently, it is advisable that steps be taken to prevent the loss of information stored on the tape.

The first step is to clean the tape head and capstan as discussed in the Installation, Operation, and Test Manual. If this does not alleviate the problem, the next step is to transfer the information to a new medium, retiring the worn tape. Continued use could cause loss of information or damage to the tape drive itself.

ERROR 81 can occur when either the tape drive or the cartridge itself fails. To determine the source of the problem, a different cartridge can be inserted. If ERROR 81 stops occurring, assume the tape itself is bad and replace it. If ERROR 81 continues to occur, the drive itself is bad. In this case, call your HP Sales and Service Office for assistance.

Optimizing Tape Use

The tape cartridge used with a Series 9800 Desktop Computer has two tracks with 426 records on each track. Records are numbered consecutively; record 0 and record 426 are both at the same end of the tape, on different tracks. Thus, records 425 and 426 are at opposite ends of the tape. This can cause a situation in which one file spans two tracks, making access time-consuming and wearing to the tape.



To avoid this situation, you can create a dummy file in record number 425, making it impossible for one file to span two tracks. The following set of operations can be used on a tape with no files on it to create this dummy file.

```
10    !   This program should be used on an unused tape
20    CREATE "A",420
30    CREATE "DUMMY",1
40    PURGE "A"
50    !   The file 'DUMMY' stays on the tape as a place
          holder
60    END
```


The file DUMMY will be in record number 425; the first five records on the tape are used by the directory which is why file A is created with only 420 records.

Chapter 12
Debugging and Error Testing

Computer Museum

page 210 • **TRACE** (monitors branching in all or part of a program)
page 211 • **TRACE WAIT** (causes a delay after each statement that causes output)
page 211 • **TRACE PAUSE** (pauses the program at a specified line)
page 212 • **TRACE VARIABLES** (monitors changes in the values of one to five variables)
page 212 • **TRACE ALL VARIABLES** (monitors all variables in all or part of program)
page 213 • **TRACE ALL** (monitors all branching and variable changes)
page 213 • **NORMAL** (cancels all tracing operations)
page 213 • **ON ERROR** (causes branching when an error occurs during program execution)
page 215 • **OFF ERROR** (cancels the ON ERROR condition)

Error Functions

ERRL (line number of last error)
ERRN (number of last error)
ERRM$ (message of last error)

Debugging a Program

Tracing a program is a convenient method of debugging the logic errors in the program. There are two types of tracing statements – three which trace the logic flow (TRACE, TRACE PAUSE, TRACE ALL) and three that trace variable assignments (TRACE VARIABLES, TRACE ALL VARIABLES, TRACE ALL of a running program. Only one of each type can be in effect at one time. A subsequent one cancels the previous one. Output from TRACE operations goes to the system comments line. When tracing, it is advisable to set the print all mode (press and latch PRT ALL) and specify a printer other than the CRT as the print all printer (with PRINT ALL IS) so that TRACE outputs are more permanent.

Tracing statements can be programmed or executed from the keyboard. They do not increase program Read/Write Memory requirements when executed from the keyboard or from the program. If you are tracing a program and execute a GET operation, the tracing will be cancelled.

Tracing operations cause the computer to temporarily revert to SERIAL mode even if OVERLAP is in effect. TRACE, TRACE ALL, and TRACE VARIABLES also slow down program execution.

The TRACE Statement

The TRACE statement is used to trace program logic flow in all or part of a program. When any branching occurs in a program, both the line number of the line where the branch is from, and the line number of the line where the branch is to are output.

TRACE [beginning line identifier [, ending line identifier]]

When a branch occurs, the output is –

TRACE---FROM line # TO line #

If no line identifiers are specified, all branches in the program are monitored. When one line identifier is specified, tracing doesn't begin until that line is executed. When both line identifiers are specified, tracing begins when the first line you specify is executed and continues (regardless of where the program is executing), then stops when the second line you specify is executed.

The TRACE WAIT Statement

The TRACE WAIT statement is used in conjunction with any other TRACE statement. It causes a specified delay to occur after each statement which causes a trace output. It is useful for monitoring and examining trace output as it occurs.

TRACE WAIT number of milliseconds

The delay is specified by a numeric expression in the range −32 768 through 32 767 which indicates the number of milliseconds after each trace printout. A negative number defaults to zero.

The TRACE PAUSE Statement

To check whether or not a line in a program is reached, or to monitor the number of times a specified line is executed, use the TRACE PAUSE statement.

TRACE PAUSE [line identifier [, numeric expression]]

If no parameters are specified, execution pauses when this statement is executed; the next line to be executed is then displayed. This allows you to pause a running program and know where it is paused, which is not possible with the PAUSE statement.

When only the line identifier is specified, the running program stops when execution reaches the specified line, but **before** the line is executed. When the numeric expression is specified, it is rounded to an integer, call it N. The program stops when the specified line is reached for the Nth time; the line isn't executed. Execution can be resumed with that line by pressing [CONT]. Every subsequent time the line is encountered, the program pauses before the line is executed.

This type of tracing can be disabled by letting the line identifier be one that is not a line identifier in memory. The most efficient way is to let it be a lower number than the lowest numbered line in memory.

The TRACE VARIABLES Statement

To trace changes in values of variables without using an output statement, use the TRACE VARIABLES statement.

TRACE VARIABLES variable list

The variable list can contain simple numeric and string variables, and array identifiers; there can be one to five items separated by commas. The value of any variable which changes is printed. The output is –

TRACE--LINE line number, variable name [(subscripts)]= value

The line number is the line in which the change occurred. If the change comes from a live keyboard operation, the line number is replaced by KEYBOARD. The new value of the variable is indicated. In the case of an array, the values of the subscript(s) at the time are printed following the name.

When an entire array changes value, the printout is –

TRACE--LINE line number, array name (*) CHANGED VALUE

Tracing variables also detects changes in subprograms of variables passed by reference. For example, suppose –

```
TRACE VARIABLES A,B
```

is executed and A is passed by reference to a subprogram. If the corresponding variable in the subprogram is changed, a trace message for A occurs.

The TRACE ALL VARIABLES Statement

To trace all variables with the ability to specify lines, use the TRACE ALL VARIABLES statement.

TRACE ALL VARIABLES [beginning line identifier [, ending line identifier]]

When no line identifiers are specified, all variables are traced throughout the program. When one line identifier is specified, tracing begins after that line is executed. The ending line identifier causes tracing to stop after that line is executed.

TRACE ALL VARIABLES cancels and is cancelled by TRACE VARIABLES.

This method of tracing can be turned off by letting the first line identifier be a line identifier which is not in memory such as an undefined label or line number which is lower than the lowest line number in memory.

The TRACE ALL Statement

To trace both program logic and variables, use the TRACE ALL statement. This statement allows, in effect, concurrent execution of TRACE and TRACE ALL VARIABLES.

```
TRACE ALL
```

Either 'part' of the TRACE ALL mode can be altered without cancelling the other part. For example, if TRACE VARIABLES A, B is executed after TRACE ALL, tracing of all variables is cancelled, and only A and B are traced, but the TRACE part of TRACE ALL is not affected.

Although the volume of printout is high, TRACE ALL is useful if a logic problem in a program hasn't been isolated with selective tracing.

The NORMAL Statement

All tracing statements are cancelled by executing SCRATCH, SCRATCH A, SCRATCH C, SCRATCH V, SCRATCH P, or the NORMAL statement –

```
NORMAL
```

Error Testing and Recovery

The ON ERROR Statement

Run-time errors are those which occur only when a program is running. Dividing by zero is an example. A run-time error normally halts execution. Through use of the ON ERROR statement, run-time errors can be caught when they occur and execution can continue with the specified line. The ON ERROR statement specifies a branching which takes place after an error occurs. Interrupt branching and how ON ERROR relates to other interrupts are covered in Chapter 14.

ON ERROR GOTO line identifier
ON ERROR GOSUB line identifier
ON ERROR CALL subprogram name[1]

1 Can't pass parameters

An ON ERROR...CALL statement is active in the program segment where it is declared and in all program segments called by that segment. ON ERROR...GOTO or GOSUB is active only in the program segment where it is declared. Execution of another ON ERROR statement cancels the previous one.

When a run-time error occurs and the ON ERROR condition has been established, execution is transferred to the specified line. Then the ERRN, ERRL, and ERRM$ functions (discussed next) could be tested, error recovery procedures or "DISP ERRM$" could be executed.

NOTE

When a program is running in OVERLAP mode, ON ERROR won't trap most I/O errors (54-103). It is advisable to use SERIAL mode when trapping errors with ON ERROR.

If GOTO is specified and the recovery routine contains an error, it is possible to program into an endless loop. It can be stopped by pressing [PAUSE] or [STOP]. If GOSUB or CALL is specified and the routine contains an error, the normal error message is displayed and execution stops.

If the ON ERROR statement specifies a GOSUB or CALL, computer priority is set at the highest level until RETURN or SUBEXIT is executed. This means that the routine can only be interrupted by an ON END# which is declared in the same program segment as ON ERROR...GOSUB, or is in the subprogram called by ON ERROR...CALL.

A routine accessed with GOTO can be interrupted because system priority isn't changed. Please refer to Chapter 14 for more information about interrupts.

Error Functions

One string and two numeric functions can be used with ON ERROR.

ERRL	The error line function returns the line number in which the most recent program execution error occurred.
ERRN	The error number function returns the number of the most recent program execution error.
ERRM$	The error message string returns the most recent program execution error message, a combination of ERRL and ERRN.

The OFF ERROR Statement

ON ERROR is disabled with the OFF ERROR statement –

```
OFF ERROR
```

Example

```
10    !   This program uses ON ERROR to detect various
          mass storage errors and display an appropriate
20    !   message if one occurs
30    ON ERROR GOTO Recovery1
40    Try=1
50 Print:     CREATE "DATA",25
60            ASSIGN #1 TO "DATA"
70            FOR I=1 TO 50
80               PRINT #1;RND
90            NEXT I
100           PRINT "DATA PRINTED ON FILE"
110           STOP
120 Recovery1:      ! Check for errors 80, 83, and 89
130           IF ERRN=80 THEN E80
140           IF ERRN=83 THEN E83
150           IF ERRN=89 THEN E89
160           GOTO Exit      ! Error was none of the above
170 E80:      PRINT "PUT THE TAPE IN.  THEN PRESS CONTINUE"
180           BEEP
190           PAUSE
200           GOTO Print
210 E83:      PRINT "TAPE IS WRITE PROTECTED. "
220           PRINT "UNPROTECT THE TAPE, THEN PRESS CONTINUE"
230           BEEP
240           PAUSE
250           GOTO Print
260 E89:      IF Try<3 THEN Print      ! 3 tries at writing
270           Try=Try+1
280           PRINT "CAN'T WRITE ON TAPE DUE TO CHECK READ ERROR"
290           PRINT "INSERT ANOTHER TAPE AND PRESS CONTINUE"
300           BEEP
310           PAUSE
320           GOTO Print
330 Exit:     PRINT ERRM$      ! Another error occurred
340           BEEP
350           STOP
```

Some errors are not trappable using the ON ERROR statement. These errors include –

- Syntax errors (not run-time errors)
- Errors occurring when statements are executed via live keyboard
- Errors detected by the I/O processor when in OVERLAP mode
- Mass storage operation errors using LOAD, STORE, GET, SAVE, LINK, LOAD ALL, STORE ALL, STORE BIN, LOAD BIN, RE-SAVE and RE-STORE.

The following errors are not trappable –

1	2	5	14
16	40	41	42
55	56	57	58
83			

Some of the errors which cannot be trapped in the OVERLAP mode can be trapped if you run the program in SERIAL mode.

Some of the mass storage errors (such as Error 80) can be trapped by doing an ASSIGN statement for a known data file on the medium, as shown here –

```
ASSIGN #1 TO "CHAMP"
```

could trap an Error 80 if the file CHAMP exists on that particular medium and it is not installed into its drive.

The ASSIGN statement can be deactivated by –

```
ASSIGN #1 TO *
```

after the ON ERROR is tested.

Chapter 13
Special Function Keys

The Special Function Keys (SFK's), marked k0 through k15, provide a variety of uses: typing aids for frequently used statements, commands, operations and other series of keystrokes, program interrupting capability and accessing CRT special features.

page 220 • **EDIT KEY** (lets you define a Special Function Key as a typing aid of up to 78 keystrokes)
page 225 • **SCRATCH KEY** (erases the typing-aid definitions of one or all Special Function Keys)
page 225 • **LIST KEY** (lists the typing aid definitions of one or all Special Function Keys)

CRT Special Features

To access the CRT special features, hold down CONTROL, then press any of these

k0 (inverse video)
k1 (blinking)
k2 (underline)

Repeat the procedure to clear the mode.

Pre-defined Definitions

These keys have the following definitions at power on or after SCRATCH A is executed.

Key	Function
CONTROL k0	Inverse video mode
CONTROL k1	Blinking mode
CONTROL k2	Underline mode
k6	CLEAR LINE REWIND":T14" EXEC (if optional tape drive present)
k7	CLEAR LINE REWIND":T15" EXEC
k8	CLEAR LINE GET
k9	CLEAR LINE LOAD
k10	CLEAR LINE SAVE
k11	CLEAR LINE STORE
k12	CLEAR LINE EDIT
k13	CLEAR LINE EDIT LINE
k14	CLEAR LINE LIST
k15	CLEAR LINE SCRATCH

Special Features

The CRT special features – inverse video, blinking and underline – can be used alone or combined. Each mode is entered by holding down CONTROL, then pressing the specific key.

Example

For an example of blinking, hold down CONTROL, then press k1

Now type in ****!!!!.

To add inverse video to blinking, hold down CONTROL, then press k0

and type in ####.

Each mode is exited by pressing [CONTROL] and the specific key again or by pressing any of the CLEAR keys. To get back to normal mode in the previous example, hold down [CONTROL] and press [k0], [k1]

These special features are very useful for highlighting text which is output to the CRT in programs. Strings can be displayed or printed to the CRT with any combination of the special features.

Example

This example shows underline.

```
10    DIM A$[25]
20    A$="DO IT NOW!!!"     ! CONT'L & K2 were pressed
                              before 'N' was typed and
30                          ! after 'W' was typed.
40                          ! This turns underline on and off
                              Be sure quotes aren't underlined
50    PRINT A$
60    !    Blinking, inverse video and underline can
70    !    emphasize text.  They can be used separately
80    !    or combined.
90    END
```

```
DO IT NOW!!!
```

Note

Entering any combination of special feature modes adds one character to the length of the string, as does exiting the special features mode. For example, the length of –

"Δ[1]12345678"

is 11 characters.

If you use a special feature in an output or REM statement, or in a comment, make sure you enter at least one character that has no special features before storing the line. Otherwise, the entire program is in that mode after that point in the program when listed.

[1] Δ specifies a blank which is a character.

Typing Aids

Keys 6 through 15 are defined at power on and SCRATCH A as typing aids so that frequently used operations can be entered with a single key stroke. These definitions are indicated below the appropriate key.

Hint

If you press a defined SFK and get an unexpected UNDEFINED KEY message, check the shift lock key.

You can save your key definitions for later use with the STORE KEY statement and retrieve them with LOAD KEY. These two statements are discussed in the mass storage chapter.

The EDIT KEY Command

There are 32 special function keys – 16 unshifted, 16 shifted – available to be defined as typing aids. The initial definitions of keys 6 through 15 are not permanent, but can be edited, or erased and redefined. These definitions were listed previously in this chapter.

NOTE

The CRT special feature definitions are permanent. They are separate from the typing-aid definitions.

To define or edit a key, execute –

EDIT KEY key number

or type in –

EDIT

(Key 12 can be used if it still has its power-on definition)

and press the key to be defined.

The computer is now in the edit key mode with the key number displayed at the top of the CRT and any current definition displayed. Any keys on the keyboard, **up to 78 keystrokes**, can be entered to define a particular key, with these exceptions –

In addition, the SFK itself may not be used in its own definition; this would cause an endless recursion.

Pressing the SFK that is being defined a second time stores the keystrokes as the definition and returns the computer to the normal mode. Pressing it immediately after the edit key mode was entered defines that key as null if that key had no previous definition. [STOP] can be pressed at any time to abort the editing of the key; no new definition is stored and any previous definition remains.

Most of the examples that follow relate to and build upon each other.

Computer Museum

Examples

For example, let's say you are keying in a program that has many PRINT statements. It would be handy to define key 0 as PRINT. Key in EDIT.

Then press [k0].

and type in PRINTΔ[1].

To store the definition, press [k0].

Now if you wanted to type in: PRINT X,Y, four keystrokes can accomplish this: [k0] [X] [,] [Y].

One SFK definition can be used to define another. For example, say that [k1] is defined Pay.

Key 2 could by defined to be PRINT Pay,H by entering the edit key mode for key 2, then pressing [k0] [k1], [H]

then storing the definition by pressing [k2].

Pressing [k2] now enters PRINT Pay,H.

---

1 Δ Indicates a blank space

An SFK can also be defined so that it performs an operation immediately. This is accomplished by having the last entry in the definition be one of the special terminator keys –

STORE CONT EXEC RUN PAUSE STEP INS LN DEL LN

Only one of these keys can be used in a key definition and it must be the last entry in the definition. A terminator key can be the 79th keystroke used to define an SFK.

Examples

In the previous example, k2 is defined as PRINT Pay,H. To define k3 as an immediate-execute operation to execute PRINT Pay,H, enter the edit key mode for key 3. Then press k2 EXEC.

Then store the definition by pressing k3.

Now when you press k3, the values of Pay and H are automatically printed.

As another example, say that you are writing a program which uses the above variables Pay and H and you want the values printed many times throughout the program. By defining k4 to be –

k2 STORE

the entire line PRINT Pay,H can be automatically stored after a line number by pressing key 4 following the line number.

If two or more SFKs that each contain a terminator key as part of their definition are used to define another SFK, execution stops with the first terminator key.

Example

For example, suppose key 16, key 17 and key 18 are defined as follows –

```
KEY 16
  PRINT "K16"
  -Execute
KEY 17
  PRINT "K17"
  -Execute
KEY 18
  -Key 16
  -Key 17
```

Pressing key 18 results in –

```
K16
```

The character editing keys [←] [→] [INS CHR] [DEL CHR] can be used to edit an SFK definition, or can be entered as part of an SFK definition. They must be pressed while [CONTROL] is held down to be entered as part of the key definition.

Examples

For example, to change the previous definition of [k2], PRINT Pay,H to DISP Pay,H, first enter the edit key mode for [k0] which was defined as PRINT.

```
PRINT _
```

is displayed. Press [→] six times to position the cursor under the P. Now type in DISP.

To delete the T, press [DEL CHR].

To store the new definition, press [k0].

The definition of key 2 is automatically altered because key 0 is part of its definition.

Here's an example of using [INS CHR] in a key definition. The definition of key 0, DISPΔ, can be changed to include quote marks and an insert cursor so that only the text need be entered into the display statement. Enter the edit key mode for key 0. Key in two quote marks, then hold down [CONTROL] and press [←] and [INS CHR]. Now press key 0 to store the definition.

Now you can press key 0, type in the text you wish to display, and execute or store the line.

Many of the keys on the keyboard, such as [TAB], [RECALL], [↑] and [EXEC] do not have a directly printable character, but instead cause some action to occur when pressed. To represent these keys in the edit key mode, each key has a unique keycode that indicates its action and is displayed on a separate line.

When any of these keys is pressed for part of an SFK definition, the previous parts of the definition roll up; the keycode for the key just pressed appears on the line above the cursor, with the cursor in the entry area ready for another key.

When editing keycodes, the four editing keys [←] [→] [DEL CHR] [INS CHR] **appear** to have a slightly different function. Using [←] to move the cursor back into previously defined parts causes the display to roll down. [→] causes it to roll up. [INS CHR] allows keystrokes to be inserted above (before) a keycode entry.

Example

For example, let's say you wanted to define [k10] to set three tabs each three spaces apart but defined it to be –

```
-Tab Clear
-Right arrow
-Right arrow
-Right arrow
-Tab set
-Right arrow
-Right arrow
-Right arrow
-Right arrow
-Tab set
```


To change the `Tab clear` to `Tab set` and delete one of the last four `Right arrow`s, do the following –

Enter the edit key mode for [k10]. The flashing cursor will be in the line under the last `Tab set`. Now press –

[→]

ten times to position `Tab Clear` in the cursor line. Now press –

[TAB SET]

To delete a `Right arrow`, press –

[→]

four times to position a `Right arrow` in the cursor line. Now press –

[DEL CHR]

to delete that entry.

Finally, press [k10].

The SCRATCH KEY Command

To erase a specified key definition, type in –

```
SCRATCH
```

(or press [k15] if it still has its power-on definition) then press the key you wish to erase.

To erase the typing-aid definitions of all special function keys, execute –

```
SCRATCH KEY
```

Erasing all SFK definitions adds 160 bytes (138 bytes if your System 45 has no lefthand tape drive) to the power-on value of space available in Read/Write Memory, since the initial SFK definitions use 160 (138) bytes.

The LIST KEY Command

All or selected SFK typing-aid definitions can be listed. Executing this command –

```
LIST KEY
```

causes all typing-aid definitions to be listed on the standard printer. (see PRINTER IS, Chapter 10). To specify a different device on which the listing is to occur, execute –

LIST KEY# select code [, HP-IB device address]

A single key can be listed by executing –

LIST KEY [#select code [, HP-IB device address];] SFK number
or
LIST [kn]

Here are some examples of LIST KEY commands –

```
LIST KEY       ! Lists all keys
LIST KEY #6    ! Lists all keys to select code 6
LIST KEY #6,2  ! Lists all keys - HP-IB printer
LIST KEY #6;8  ! Lists key #8 to select code 6
LIST KEY 8     ! Lists key number 8
```

13

Chapter 14
Program Interrupts

Introduction

Your computer has five interrupt declarative statements that enable program interrupt conditions to be specified in a program. The specified interrupt conditions cause a GOTO, GOSUB or CALL branching to occur. The five interrupt statements allow interrupts to come from –

- a Special Function Key (ON KEY)
- any key on the keyboard (ON KBD)
- a program error (ON ERROR)
- an end-of-file condition (ON END)
- a peripheral device (ON INT)

This chapter covers ON KEY, ON KBD and general information about the five statements and how they are related. ON ERROR is covered in Chapter 12, ON END in Chapter 11 and ON INT in the I/O ROM Manual.

Any program interrupt, be it a keystroke, error, end-of-file, or peripheral interrupt, occurs while some line of the program is executing. Branching that is enabled by an interrupt declarative occurs when the program line is complete and is known as an **end-of-line branch**. Thus, more than one interrupt can occur during statements that take a long time to execute like WAIT or MAT...INV. These are known as **simultaneous interrupts**.

Priority

Priority determines whether or not a program can be interrupted and also in what order simultaneous interrupts are handled.

At power on, the priority of the computer is set to 0. All operations then assume this priority. The ON declaratives specify a higher priority so that a program can be interrupted. A routine can only interrupt the program if it has a higher priority than the current priority.

In the ON KEY, ON KBD and ON INT statements, you can specify priority in the range 1 through 15. If you don't specify it, it defaults to 1.

The ON END and ON ERROR statements have an implied priority of 16. Thus, they can always interrupt a program.

Changing System Priority

When an interrupt is declared with CALL or GOSUB, system priority is set to the specified priority when the branching occurs. When GOTO is specified, system priority is not changed. Since GOTO doesn't change system priority, another lower priority interrupt can interrupt as soon as the line specified by GOTO is executed. Thus, you should use GOTO in an interrupt only if you don't care if the program doesn't return to where it was when the interrupt occurred or if the interrupt routine is interrupted.

When the routine entered with CALL or GOSUB is exited system priority is set back to what is was before the interrupt.

Scope of Interrupt Statements

An interrupt declarative with CALL is active within the program segment it is declared in and within any subprogram called by that segment. An interrupt declarative with GOTO or GOSUB is only active within the particular program segment. Branching to a subprogram suspends GOTO and GOSUB interrupts until the program exits back to the program segment in which they were defined. Interrupts relating to those GOTO and GOSUB interrupts are logged (one for each declarative), then processed upon return.

How Interrupts Interact

Interrupt declaratives can be split into two groups: one containing ON KEY, ON KBD and ON INT, the other containing ON END and ON ERROR. This distinction is made because priority for the latter two is always implicitly the highest; for the former group, it can be specified.

ON KEY, ON KBD and ON INT

Here are some facts about ON KEY, ON KBD and ON INT interrupts –

- GOSUB interrupt routines can't be entered again with a repeat of their interrupt until RETURN is executed.
- Another interrupt with a higher priority can interrupt an interrupt routine in progress. The lower-priority routine is completed when the higher one is done if the higher one specified GOSUB or CALL. A lower-priority interrupt can't interrupt a higher-priority one; it is executed when the higher is done.

- If simultaneous interrupts with the same priority are recorded, ON KBD takes precedence over ON INT and ON INT takes precedence over ON KEY. Multiple ON KEY interrupts with the same priority are handled in descending key number order. Multiple ON INT interrupts with the same priority are handled in descending select code order.
- To activate ON KEY declarations after ON KBD has been executed, OFF KBD must be executed.
- If ON KBD is cancelled with OFF KBD while an ON KBD interrupt routine is executing, any ON KEY with a high enough priority can interrupt the ON KBD interrupt routine.

ON ERROR and ON END

ON ERROR and ON END have an implied priority of 16, meaning they can interrupt at any time. Here are some facts about how they are related –

- An ON ERROR routine can interrupt an ON END routine if they are both declared in the same program segment or if ON ERROR is declared with CALL in a subprogram which also calls the ON END routine.
- An ON END routine can interrupt an ON ERROR routine if they are declared in the same program segment. If ON ERROR specifies CALL, ON END can only interrupt the error subprogram if ASSIGN and ON END are re-specified in the subprogram. This must be done since file assignments in a calling program aren't active when a subprogram is called.

Errors

If the line or subprogram specified in ON KEY, ON KBD or ON KEY doesn't exist, the specified interrupt causes an error when it occurs. The line number in the error message won't be the line number of the ON statement, but will be the line number of the line that was executing when the interrupt occurred.

When are Interrupts Active?

At power on and SCRATCH A, all interrupt statements are activated. You can suspend interrupts using the DISABLE statement.

The DISABLE Statement

Any ON KEY, ON KBD and ON INT declaratives in any program segment are suspended by executing the DISABLE statement –

```
DISABLE
```

One ON KEY interrupt per key, up to 80 keystrokes for ON KBD and one ON INT interrupt per select code can be logged, but the interrupt routines are not executed until declaratives are reactivated. Then interrupts are serviced according to priority.

The ENABLE Statement

ON KEY and ON INT declaratives are reactivated by executing the ENABLE statement –

```
ENABLE
```


The ON KBD Statement

The ON KBD (on keyboard) statement allows the keyboard to be used like an external input device, operating on an interrupt service level. It is used to keep track of which keys on the keyboard are pressed. This is useful in terminal emulator applications and applications where you want to override the normal operation of a key or keys. When ON KBD is executed, the **peripheral keyboard mode** is set. This also disables live keyboard mode and any ON KEY statements.

ON KBD [priority] GOTO or GOSUB line identifier [, ALL]
ON KBD [priority] CALL subprogram name[1] [, ALL]

All keys that are pressed are logged into an ON KBD buffer except the following –

[↑], [↓], [ROLL ↑], [ROLL ↓] [TYPWTR], [STOP], [CONTROL]-[STOP] (reset), [AUTOST] and [PRT ALL]

The ON KBD statement can be executed only within a main program, not within a subprogram or from the keyboard. If you put an ON KBD statement in a subprogram, ERROR 104 occurs. Executing another ON KBD statement cancels the previous one.

1 Can't pass parameters.

Priority

Priority determines when interrupts are handled. An interrupt can only interrupt the program if its priority is higher than the current system priority. The priority parameter is a numeric expression in the range 1 through 15. If it is not specified, 1 is used as the default value. GOSUB and CALL set system priority to the specified level. GOTO leaves system priority unchanged.

ALL

The ALL parameter specifies that all keycodes are trapped except [CONTROL]-[STOP] (reset), [AUTO ST], and [PRT ALL].

Computer Museum

ON KBD Buffer

When an ON KBD statement is executed, an 80-keystroke buffer is established to hold key codes for the keys that are pressed. When a key is pressed, its keycode is placed into the buffer.

When the buffer is full and a key is pressed, the computer beeps to advise you of this.

The buffer is emptied every time the KBD$ function (discussed later in this section) is used.

Considerations

Here are some facts that should be taken into account when using ON KBD –

- DISABLE and ENABLE suspend and reactivate ON KBD. Up to 80 keystrokes are logged for later processing when DISABLE is in effect.
- A PAUSE, TRACE PAUSE or STOP statement in the program cancels the ON KBD condition.
- Any type of INPUT or EDIT statement temporarily disables ON KBD. The values input do not go into the KBD buffer. The values in the keyboard buffer are saved and restored when the input is complete.
- ON KBD has priority over any ON KEY statement.

KBD$ Function

The KBD$ function returns the entire contents of the buffer set up by ON KBD. Its maximum length is 80 characters. Every time KBD$ is referenced in a statement, the current buffer contents are put into KBD$ and the buffer is emptied. Thus, if you want to process the contents of the buffer, the first reference to KBD$ **must** store the contents of KBD$ into another string variable.

The null string is returned as the value of KBD$ when the buffer is empty or when peripheral keyboard mode is no longer set because OFF KBD was executed.

For all ASCII keys pressed, the value in KBD$ is that character (See the Reference Tables for ASCII characters). For non-ASCII keys ([•] or [k1] for example), the keycode should be interpreted using the NUM function to get a meaningful numeric interpretation of the keycode. For non-ASCII keys, two values are returned. The first is 255 which indicates non-ASCII. The second value is indicated in the following table. If [SHIFT] is held down when the key is pressed, 64 is added to the value. If [CONTROL] is held down, 128 is added to the value.

Key	Decimal Value	Key	Decimal Value
SFK #0	0	LEFT ARROW	22
SFK #1	1	RIGHT ARROW	23
SFK #2	2	UP ARROW	24
SFK #3	3	DOWN ARROW	25
SFK #4	4	ROLL UP	26
SFK #5	5	ROLL DOWN	27
SFK #6	6	HOME	28
SFK #7	7	CLEAR	29
SFK #8	8	CLEAR TO END	30
SFK #9	9	DELETE CHAR	31
SFK #10	10	INSERT CHARACTER	32
SFK #11	11	DELETE LINE	33
SFK #12	12	INSERT LINE	34
SFK #13	13	RECALL	35
SFK #14	14	TAB	36
SFK #15	15	TAB SET	37
STEP	16	TAB CLEAR	38
PAUSE	17	TYPEWRITER	39
RUN	18	BACKSPACE	50
CONT	19	RESult	51
STORE	20	STOP	52
EXECUTE	21	CLEAR LINE	53

Examples

Here are some example uses of ON KBD and KBD$ –

```
10      ! This program traps keystrokes, determines ASCII or
          non-ASCII and prints its keycode
20      DIM Keys$[80]
30      Stopcode=52
40      ON KBD GOSUB Service_kbd ,ALL    ! Let program trap keys
50      DISP "PRESS ANY KEYS"
60      GOTO 50
70 !
80 Service_kbd: Keys$=KBD$      ! Save all keys now in ONKBD buffer.
90              WAIT 1000               ! Accumulate keys
100             FOR I=1 TO LEN(Keys$) ! Process all keys.
110             K$=Keys$[I;1]
120             IF NUM(K$)=255 THEN Non_ascii
130             PRINT "ASCII key = ";K$;"; keycode is";NUM(K$)
140             GOTO Next_key
150 Non_ascii:  I=I+1           ! Skip over the non-ASCII indicator.
160             K$=Keys$[I;1]
170             PRINT "Non-ASCII key; keycode is";NUM(K$)
180             IF NUM(K$)=Stopcode THEN STOP  ! Trap STOP
190 Next_key:   NEXT I          ! Advance to the next key.
200             RETURN          ! Done processing keys.
```

```
ASCII key = T; keycode is 84
ASCII key = R; keycode is 82
ASCII key = A; keycode is 65
ASCII key = P; keycode is 80
ASCII key = P; keycode is 80
ASCII key = I; keycode is 73
ASCII key = N; keycode is 78
ASCII key = G; keycode is 71
ASCII key =  ; keycode is 32
ASCII key = K; keycode is 75
ASCII key = E; keycode is 69
ASCII key = Y; keycode is 89
ASCII key = S; keycode is 83
ASCII key =  ; keycode is 32
Non-ASCII key; keycode is 52
```

The OFF KBD Statement

The OFF KBD statement cancels a previously executed ON KBD statement and peripheral keyboard mode, thus allowing ON KEY and live keyboard to be active. It also clears the ON KBD buffer.

```
OFF KBD
```

The ON KEY# Statement

The 32 special function keys can be used to interrupt a running program and cause branching. This branching capability is useful for a program which requires user intervention. Each key can be defined to cause a specific branch, so that the user can steer the program the way he wants it. For example, a 'menu' of various routines can be displayed and accessed using special function keys. Here is where a blank key overlay can be used.

This interrupt capability is declared with an ON KEY# statement which specifies the branching operation and the related SFK.

ON KEY# key number [, priority] GOTO or GOSUB line identifier
ON KEY# key number [, priority] CALL subprogram name[1]

The key number is an integer in the range 0 through 31. When a key is pressed and an ON KEY# has been declared for it, the specified branching occurs if the specified priority exceeds current system priority. System priority remains unchanged if GOTO is specified and is changed to the indicated priority if GOSUB or CALL is specified.

Priority

The priority determines the order in which multiple interrupts are handled. The range of priority is 1 through 15. If it is not specified, it is assumed to be 1. An interrupt routine can only interrupt the program if it has a higher priority than the current system priority.

1 Parameters can't be passed.

Example

Here's an example that illustrates ON KEY# and priority –

```
10   !   *************** INTERRUPT WITH KEYS ***************
                         PRESS KEYS 4,3,2 AND 1
20   ON KEY #1,4 GOSUB Boss          ! Highest priority
30   ON KEY #2,3 GOSUB Phone         ! 2nd highest priority
40   ON KEY #3,2 GOSUB Spouse        ! 3rd highest priority
50   ON KEY #4,1 GOSUB Coffee        ! Lowest priority
60   PRINTER IS 16
70   PRINT SPA(20);"ON KEY#1,4 GOSUB Boss"
80   PRINT SPA(20);"ON KEY#2,3 GOSUB Phone"
90   PRINT SPA(20);"ON KEY#3,2 GOSUB Spouse"
100  PRINT SPA(20);"ON KEY#4,1 GOSUB Coffee"
110  PRINT LIN(5)
120  DISP "WAITING  -- PRESS KEYS 4,3,2,1"
130  GOTO 120
140  STOP
150  !   **************** INTERRUPT ROUTINES *****************
160 Boss:     FOR I=1 TO 10
170           PRINT "KEY 1";I;TAB(15);"Meet with the boss"
180           NEXT I
190           RETURN
200 Phone:    FOR J=1 TO 10
210           PRINT "KEY 2";J;TAB(15);"Talk on the phone"
220           NEXT J
230           RETURN
240 Spouse:   FOR K=1 TO 10
250           PRINT "KEY 3";K;TAB(15);"Call spouse"
260           NEXT K
270           RETURN
280 Coffee:   FOR L=1 TO 10
290           PRINT "KEY 4";L;TAB(15);"Drink coffee"
300           NEXT L
310           RETURN
320  END
```

```
                    ON KEY#1,4 GOSUB Boss
                    ON KEY#2,3 GOSUB Phone
                    ON KEY#3,2 GOSUB Spouse
                    ON KEY#4,1 GOSUB Coffee


KEY 4 1        Drink coffee
KEY 3 1        Call spouse
KEY 2 1        Talk on the phone
KEY 1 1        Meet with the boss
KEY 1 2        Meet with the boss
KEY 1 3        Meet with the boss
KEY 1 4        Meet with the boss
KEY 1 5        Meet with the boss
KEY 1 6        Meet with the boss
KEY 1 7        Meet with the boss
KEY 1 8        Meet with the boss
KEY 1 9        Meet with the boss
KEY 1 10       Meet with the boss
KEY 2 2        Talk on the phone
KEY 2 3        Talk on the phone
KEY 2 4        Talk on the phone
KEY 2 5        Talk on the phone
KEY 2 6        Talk on the phone
KEY 2 7        Talk on the phone
KEY 2 8        Talk on the phone
KEY 2 9        Talk on the phone
KEY 2 10       Talk on the phone
KEY 3 2        Call spouse
KEY 3 3        Call spouse
KEY 3 4        Call spouse
KEY 3 5        Call spouse
KEY 3 6        Call spouse
KEY 3 7        Call spouse
KEY 3 8        Call spouse
KEY 3 9        Call spouse
KEY 3 10       Call spouse
KEY 4 2        Drink coffee
KEY 4 3        Drink coffee
KEY 4 4        Drink coffee
KEY 4 5        Drink coffee
KEY 4 6        Drink coffee
KEY 4 7        Drink coffee
KEY 4 8        Drink coffee
KEY 4 9        Drink coffee
KEY 4 10       Drink coffee
```

If multiple ON KEY# declaratives have the same priority, the declarative with the highest key number is given preference when two keys are logged as simultaneous interrupts.

Considerations

ON KEY# statements which specify GOTO or GOSUB are active only in the program segment in which they were declared. A CALL interrupt is active in the program segment in which it was declared and in all subprograms called by that segment. ON KEY# declaratives are suspended while a program is waiting for a response to an INPUT, LINPUT or EDIT statement and after PAUSE is executed.

If an ON KEY...GOSUB or CALL routine has not been completed and that key is pressed again, the key won't be acknowledged until the first routine is completed when its RETURN or SUBEND is executed.

If a special function key has both ON KEY# and typing aid definitions, the ON KEY# has precedence while the program is running. Remember, waits caused by PAUSE, INPUT, LINPUT, and EDIT temporarily suspend the ON KEY#, so any typing aid definition is active at that time.

THE OFF KEY Statement

The ON KEY# declarative holds for a key until another declarative for the same key, SCRATCH A, SCRATCH, SCRATCH P, SCRATCH V, SCRATCH C, (RUN) or OFF KEY# is executed –

OFF KEY# key number

Summary

Here are some facts to remember when using ON KEY –

- The range of priority is 1 through 15.
- System priority is not changed when GOTO is specified.
- ON KEY declaratives are temporarily deactivated by INPUT, LINPUT, EDIT and PAUSE.
- An ON KEY declarative is permanently deactivated by another ON KEY for that particular key, SCRATCH, SCRATCH A, SCRATCH C, SCRATCH V, SCRATCH P, (RUN) or OFF KEY for the key.

Softkeys

The softkeys are used with the ON KEY statement. Complete information covering the use of Softkeys as an interactive graphics device is contained in Chapter 13 of the Color Graphics Manual.

Appendix A
Advanced Printing Techniques

Introduction

This appendix introduces you to the more advanced printing capabilities of the CRT and the internal thermal printer.

Some of the special capabilities are accessed by using various ASCII* control characters. See the ASCII table in the Reference Tables for a complete list of ASCII characters. Another capabilities allows the CRT special features: inverse video, blinking, underline, and color modes to be accessed in a program rather than using the CONTROL key. A third capability uses escape codes to address any location on the CRT selectively. The escape code sequences are compatible with those used by HP 2640-series terminals. A fourth capability uses escape codes to access capabilities of the internal printer which allow you to generate new characters replace any character, print 150%characters and more.

Many of the examples in this appendix are meant to be tried because it is impossible to show many of the CRT capabilities on the printed page.

A summary of escape code sequences can be found at the end of this appendix.

CRT Memory

Every line that is printed to the CRT is stored in the CRT memory. This memory can hold 50 80-character lines. Fewer longer lines or more shorter lines can be stored. When the memory becomes full, each new line printed to the CRT causes the oldest line in memory to be lost. All lines in CRT memory can be viewed with [↑], [↑], [ROLL↑], or [ROLL↑]. The CRT memory is cleared with [CLEAR], a formfeed character (`PRINT PAGE`, `PRINT USING @`, `PRINT CHR$(12)`) or with `ESC E`.

CRT Special Features

The special features: blinking, underline, inverse video, and color can be accessed in a program by using the CHR$ function or escape code sequence within an output statement. Any time a mode is accessed or cleared, one character is added to the length of what is output, though it is an unprinted character. This is a point to remember when dimensioning strings. Any combination of the features can be accessed by outputting –

CHR$ (n)

Where n is an integer in the range 128 through 143 and specifies which combination of features is to be accessed. 144 through 159 are equivalent to 128 through 143. 160 through 255 are for the Nationalized and Drawing characters.

* American Standard Code for Information Interchange.

The following tables show which numbers provide access to which features:

CLR	IV	BL	IV BL	UL	IV UL	BL UL	IV BL UL
128	129	130	131	132	133	134	135
144	145	146	147	148	149	150	151

White	Red	Yellow	Green	Cyan	Blue	Magenta	Black
136	137	138	139	140	141	142	143
152	153	154	155	156	157	158	159

The following escape code sequence can also be used to access the special features –

ESC&dX

X can be –

X	Result
@	CLR
A	BL
B	IV
C	IV, BL
D	UL
E	UL, BL
F	UL, IV
G	UL, IV, BL

X	Result
H	White
I	Red
J	Yellow
K	Green
L	Cyan
M	Blue
N	Magenta
O	Black

CLR - Clear IV, BL, and UL
IV - Inverse Video
BL - Blinking
UL - Underline

NOTE

The highlight features (IV, BL, and UL) are independent of the color features. Therefore, setting a highlight does not change the color and setting a color does not change the highlight.

All special features accessed with CHR$ remain in effect until specifically cleared. This can be done with the CLR feature above (for IV, BL, and UL) or by pressing the CLEAR key. Those accessed with the escape code sequence remain in effect until the end of the line or until another one is specified. At the end of a line, the highlight is cleared and the color set to white, if the escape code sequence was used.

Examples

Here are some examples to try –

```
10   PRINTER IS 16
20   PRINT "THE FOLLOWING LINES SHOW";
30   PRINT " EXAMPLES OF THE VARIOUS CRT FEATURES"
40   PRINT CHR$(135);"All three features"
50   PRINT CHR$(135);"Notice underline is dark here"
60   PRINT CHR$(133);"Inverse video, underline"
70   PRINT CHR$(132);"Underline"
80   PRINT CHR$(130);"This line blinks"
90   PRINT CHR$(128);"All features cleared"
100  END
```

Page 210:

The last sentence on the page should be:

When both line identifiers are specified, tracing begins when the first line you specify is executed and continues (regardless of where the program is executing), then stops when the second line you specify is executed.

Page 216:

The first non-trappable error should be:

- Syntax errors (not run-time errors)

Computer Museum

Page 240:

Add the following information:

9845A vs. 9845B/C

- Some of HP's software will run only on particular models of the System 45. If you have questions about software compatibility, refer to the System 45 Pricing Information brochure (P/N 5953-4572D) or call your HP Sales and Service Office.

9845C Graphics ROM vs. Enhanced Graphics ROM

Listed below are differences between the 9845C Graphics ROM (for the System 45C) and the Enhanced Graphics ROM (for the System 45B Model 200/250/290 or the System 45C):

- The Enhanced Graphics ROM provides the following capabilities which are not provided by the 9845C Graphics ROM:
  - Rubber banding (POINTER type = 4, 5, or 6)
  - Fast tracking (GRAPHICS INPUT IS TABLET)
  - Fast alpha
- The Enhanced Graphics ROM provides some capabilities that work only on a System 45B Model 2XX and not on a System 45C (refer to the Monochromatic Graphics Manual for more information):
  - Fast erasing
  - Arcs and circles
  - Rubber banding with no background loss
- Programs which have been STOREd on a computer which has the 9845C Graphics ROM can be LOADed on a computer which has the Enhanced Graphics ROM, and vice versa. The only exceptions in program execution are:

- "type" = 4, 5, or 6 in POINTER statement
- 'graphics input identifier string" = TABLET in GRAPHICS INPUT IS statement
- PLOTTER IS HPGL specifies 8 pens with Enhanced Graphics ROM and 4 pens for 9845C Graphics ROM
- GLOAD and GSTORE do not work between color and monochromatic CRTs.

- You can LOAD ALL any STORE ALL files between the System 45B Model 2XX and the System 45C, as long as both computers have the Enhanced Graphics ROM, the same options, and the same memory sizes.

- The Rubber band binary provided in the System 45C Utilities should be used only if your System 45C has the 9845C Graphics ROM. Rubberbanding is provided by the Enhanced Graphics ROM.

- The two ROMs give different results if you use alphanumeric highlights with screen addressing and overprinting in your programs.

- The following keywords have no effect when executed on a System 45B Model 2XX:
  - DEGAUSS
  - CONVERGE
  - MEMORY
  - GSTAT(5,6, or 7)

- The Monochromatic Graphics Manual explains how statements which specify colors and memory planes are executed on a System 45B Model 2XX (monochromatic CRT).

Standard Processor vs. Enhanced Processor

A peculiarity can exist when running a program in OVERLAP mode on the enhanced processor. Adding WAIT statements can make the program run faster, if they are added so that I/O statements are not executed faster than the I/O devices can handle them. In other words, I/O statements should be spaced so that a new I/O statement is issued just as the previous statement (on the same select code) completes.

How Fast is the Enhanced Processor?

The increased speed of the enhanced processor is due to "microcoding" the parts of the operating system where the language processor spends most of its time. These microcode routines run from five to fifteen times faster than the standard processor routines. The result is an overall speed increase of from three to four times, depending on the program. (The enhanced processor has little effect on assembly language programs.)

Most of the microcode routines affect the Arithmetic Logic Unit (ALU) of the language processor. Thus, speed increases are most noticeable when a program is spending almost all of its time computing and there are no I/O operations being done.

Because of differences in program structure and application, it is difficult to determine the speed increase which you may see with the enhanced processor. However, it is possible to get an idea of how much faster the individual routines run. The System 45 Computer Specifications brochure includes run times for the math and trigonometric routines (for both the standard and enhanced processors).

Maximizing Performance

There are some general guidelines to follow to make your programs run as fast as possible, regardless of the processor that your desktop computer has. Here are a few of them:

- Use full-precision (REAL) variables for math operations.
- Use integer-precision (INTEGER) variables in FOR-NEXT loops.
- Use integer-precision (INTEGER) variables for array subscripts.
- Do not use short-precision (SHORT) variables.
- Use GOSUBs instead of CALLs whenever possible.
- Turn off TRACE and TRACK when not needed.

Page 241:

The third sentence in the second paragraph should be:

Another capability allows the CRT special features: inverse video, blinking, underline, and color modes to be accessed in a program rather than using the CONTROL key.

The first sentence under "CRT Special Features" should be:

The special features: blinking, underline, inverse video, and color can be accessed in a program by using the CHR$ function or escape code sequence within an output statement.

The paragraph at the bottom of the page should read:

Where n is an integer in the range 128 through 143 and specifies which combination of features is to be accessed. 144 through 159 are equivalent to 128 through 143. 160 through 255 are for the Nationalized and Drawing characters.

The following example illustrates the differing effects of commas and semicolons –

```
10   PRINTER IS 16
20   PRINT "These lines illustrate the effects";
30   PRINT " of ; and , on field lengths."
40   PRINT "First feature is inverse video, ";
50   PRINT "second feature is underline.";LIN(1)
60   A$="**"
70   PRINT "SEMICOLON AFTER ** :"
80   PRINT "Comma after turning feature on:"
90   PRINT CHR$(129),A$;CHR$(132),A$;CHR$(128)
100  PRINT "Semicolon after turning feature on:"
110  PRINT CHR$(129);A$;CHR$(132);A$;CHR$(128)
120  PRINT LIN(2)
130  PRINT "COMMA AFTER ** ;"
140  PRINT "Comma after turning feature on:"
150  PRINT CHR$(129),A$,CHR$(132),A$,CHR$(128)
160  PRINT "Semicolon after turning feature on:"
170  PRINT CHR$(129);A$,CHR$(132);A$,CHR$(128)
180  END
```

A

Using Control Codes

ASCII characters are letters, numbers, characters and codes which each correspond to a unique 7-bit byte pattern. Each character also has equivalent decimal, binary and octal representations. The first 32 are control codes which pass control information between devices such as a carriage return or linefeed.

The control codes can be accessed for output using [CONTROL] or the CHR$ function. A two-letter symbol specifies the control code. To determine what keys can be used with [CONTROL] to obtain a control code, use the ASCII table in the Reference Tables. By following the line all the way across from the desired code, the two or three keys which produce the desired character when pressed with [CONTROL] can be determined. For example, LF (linefeed) can be obtained by pressing [CONTROL] with either [SHIFT] [* 8], [J], [SHIFT] [J], or [*]. The DEL character is the only one that can't be obtained using [CONTROL].

Basic control operations on your computer utilize five control codes which affect output to the CRT or internal thermal printer. Here are the codes and their various results –

Control Code	CRT (DISP)	CRT (PRINT)	Internal Printer
BELL	Beep	Beep	Nothing
BS(backspace)	Back up and replace	Back up and replace	Backup and replace
LF (line feed)	Nothing	Generate line feed only	Generate line feed only
FF (form feed)	Clear display line	Clear printout area and CRT memory	Search for top-of-form
CR (carriage-return)	Clear display line	Return to beginning of line	Print; roll back one line

With the exception of the control codes described above, HT (horizontal tab, CHR$(9)) and ESC (escape code, CHR$(27)) which are discussed later in this appendix; all other control codes are ignored by your computer.

A

Example

Command	Output
PRINT "ABC⌫D"	ABD
PRINT "ABC↵"	Clears CRT
PRINT "ABC↵DE"	DEC
PRINT "ABC↓DE"	ABC DE

CRT vs. Printer

There are a few cases where executing the same operation on the CRT and internal thermal printer produces differing results. These incompatiblities occur when printing ⌫ or ↵.

On the CRT, backspace causes the previous character to be cleared and replaced. On the internal printer, if the character following the ⌫ is an underline (CHR$ (95)), the previous character is not cleared.

Examples

Operation	Result-CRT	Result-Internal Printer
PRINT "A⌫B"	B	B
PRINT "A⌫ ✓ " CONTROL k2	_	_
PRINT "A⌫_"	_	A

The carriage return character also causes slightly different results on the CRT and the internal printer. On the CRT, previous characters are cleared. On the printer, they are over-printed.

Examples

Operation	Result-CRT	Result-Internal Printer
PRINT "T↵V"	V	V
PRINT "O↵/"	/	0
PRINT "A↵ ✓ " CONTROL k2	_	A
PRINT "A↵_"	_	A

Considerations

Control codes used within a BASIC statement are executed even when a program is listed. This can produce some undesirable results. For example, try listing these program lines –

```
10 PRINTER IS 16
20 PRINT " HEADING↓↓↓" !↓ is control J
30 PRINT "<=>↵///"      !↵ is control M
```

Thus, a program listing will be more readable if control codes are generated with the CHR$ function.

Disabling Control Codes

All control codes can be disabled (their action won't be performed) and viewed using the following escape code sequence –

ESC Y

The only control code which is then recognized is CR (carriage return). When one is encountered, CR is printed and a carriage return-linefeed is executed. To see how this works, output ESC Y, then list the program in the previous section.

The control codes are re-activated using CLEAR or the following escape code sequence –

ESC Z

All control features and escape code sequences are cleared, the display and printer reset, and CRT memory cleared using CLEAR or the following escape code sequence –

ESC E

If ESC Y is in effect, ESC E has no effect.

CRT Selective Addressing

Introduction

The top twenty lines of the CRT are known as the printout area. All lines printed to the CRT are stored in CRT memory which was discussed at the beginning of this appendix. Through the use of escape code sequences, any line in CRT memory can be selectively addressed and modified.

The operations available are as follows –

- Cursor Positioning
  - Absolute addressing
  - Relative addressing
  - Backspace
  - Space
  - Up
  - Down
  - Set tab
  - Clear tab
  - Tab
  - Home position – first row
  - Home position – row after last row
- Display Positioning
  - Roll up
  - Roll down
  - Next page
  - Previous page
  - Memory lock
- Editing
  - Delete line
  - Insert line
  - Clear to end of line
  - Clear to end of screen
  - Insert character
  - Delete character

Selective cursor addressing and the other operations which are covered in the rest of this appendix are very useful for form filling and text processing applications. It is recommended that you use PRINT USING with # in the format string to output the escape code sequences to avoid unexpected carriage return/linefeeds which can occur from length added to the output.

There are two example programs at the end of this appendix which combine many of the operations to manipulate output.

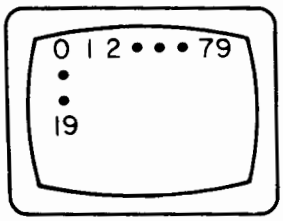
The Cursor

Any location on the screen can be addressed and a non-visible cursor specified as being there. (This cursor is **not** the same as the flashing cursor which is present in normal keyboard usage.) This cursor refers to a logical print position in CRT memory where the next character will be printed. In this appendix, the word "cursor" always refers to the logical print position.

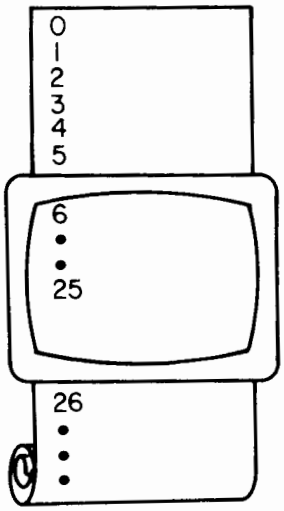
A

Addressing Schemes

The printout area is addressed using rows 0 through 19 and columns 0 through 79. The following drawing illustrates this –



CRT memory is addressed using columns 0 through 79. The number of rows depends on line length. The maximum number of lines was covered at the beginning of this appendix. The following drawing illustrates addressing of CRT memory –



In this drawing, line 6 of the CRT memory is positioned on line 0 of the printout area.

Setting the Cursor Position

The cursor can be set to any character position in the 20 lines of the printout area using absolute or relative addressing, or a combination absolute and relative addressing.

Absolute Addressing

The cursor can be set to an absolute row and column position with any of the following escape code sequences –

```
Ec&a nn r nn C
Ec&a nn c nn R
Ec&a nn y nn C
Ec&a nn c nn Y
```

Here are some guidelines for using these escape code sequences –

- nn specifies a one or two-digit number which is used to specify the row and column number. The digits preceding the R(r) specify the row number of **CRT memory**. The digits preceding the Y(y) specify the row number of the **printout area**. The digits preceding the C(c) specify the column number.
- The end of the escape code sequence is signified with a capital letter. Previous letters in the sequence are lower case.
- The first column of the printout area is addressed using 0. The maximum column address is 79; if anything greater is specified, 79 is used.
- The first row of either CRT memory or printed output is addressed using 0.
- If R is used to specify CRT memory and the specified row is not on the CRT screen, the display will roll up or down as necessary.
- When Y is used to specify lines of the printout area, the range of rows is 0 through 19. If a number < 0 is specified, line 0 is accessed. If a number > 19 is specified, line 19 is accessed.

The cursor can be moved within a row by omitting the R and preceding digits. Here is the escape code sequence –

```
Ec&a nn C
```

Similarly, the cursor can be moved within a column by omitting C and preceding digits. Here are the escape code sequences –

```
Ec&a nn R
Ec&a nn Y
```

`PRINT USING"#,K";CHR$(27)&"&a25r60C"`	Moves the cursor to row 25, column 60.
`PRINT USING"#,K";CHR$(27)&"&a60c17R"`	Moves the cursor to column 60, row 17.
`PRINT USING"#,K";CHR$(27)&"&a15R"`	Moves the cursor to row 15, current column.
`PRINT USING"#,K";CHR$(27)&"&a30C"`	Moves the cursor to column 30, current row.
`PRINT USING"#,K";CHR$(27)&"&a7Y"`	Moves the cursor to row 7 of the printout area, current column

A

Relative Addressing

The cursor can also be repositioned using relative addressing. From its current position, the cursor can be moved up (negative number) or down (positive number), left (negative number) or right (positive number). Here are the escape code sequences to use –

```
Ec&a S nn r S nn C
Ec&a S nn c S nn R
Ec&a S nn y S nn C
Ec&a S nn c S nn Y
Ec&a S nn C
Ec&a S nn R
Ec&a S nn Y
```

Here are some guidelines for using these escape code sequences –

- nn specifies a one or two digit number which is used to specify the number of rows and/or columns the cursor is to move. The digits preceding the R(r) or Y(y) specify the number of rows; the digits preceding the C(c) specify the number of columns.
- The end of the escape code sequence is specified with a capital letter. Previous letters are lowercase.
- S specifies a sign : + or –. A plus sign (+) specifies right or down. A minus sign (–) specifies left or up.
- If the number of columns specified is greater than the number of columns remaining after the cursor in the current line, the cursor is positioned in the first column (negative movement specified) or in the last column (positive movement specified). If the number of rows specified in the negative direction is greater than the current row, the cursor is positioned in the first row.

Examples

`PRINT USING"#,K";CHR$(27)&"&a+8r-10C"`	Moves the cursor down 8 rows, left 10 columns from its current position.
`PRINT USING"#,K";CHR$(27)&"&a+7c-11R"`	Moves the cursor right 7 columns, up 11 rows from its current position
`PRINT USING"#,K";CHR$(27)&"&a-8R"`	Moves the cursor up 8 rows from its current position
`PRINT USING"#,K";CHR$(27)&"&a+10C"`	Moves the cursor right 10 columns from its current position.

Combining Absolute and Relative Addressing

The cursor can be positioned to a new position using a combination of absolute and relative addressing.

Examples

`PRINT USING"#,K";CHR$(27)&"&a+8r 60C"`	Moves the cursor to column 60 and down 8 rows from its current row.
`PRINT USING"#,K";CHR$(27)&"&a-15c10R"`	Moves the cursor to row 10 and left 15 columns from its current position.

A

Moving the Cursor

The following escape code sequences can also be used to move the cursor –

ESC A	Move cursor up one row
ESC B	Move cursor down one row
ESC C	Move cursor right one column
ESC D	Move cursor left one column
ESC F	Move cursor to row after last row of CRT memory, first column
ESC G	Move cursor to first column, current row
ESC H	Moves the cursor to first row of CRT memory, first column
ESC I	Move the cursor to next set tab position.

These escape code sequences can be used very easily by defining Special Function Keys to set the cursor position, then move it up, down, left, and right.

ESC A – ESC D cause the cursor to "wrap around" when the edge of the screen is reached. When the cursor is being moved to the right it wraps around on to the next line. When the cursor is being moved to the left, it wraps around onto the previous line. ESC F can be used to return the cursor for normal printing after using cursor-moving escape code sequences. ESC H and ESC F cause the lines to scroll, if necessary.

Example

Here is an example using cursor moving to fill in blanks in a form letter.

```
10   PRINTER IS 16
20   DIM Name$[25],Magazine$[50]
30   PRINT PAGE
40   A$=CHR$(27)&"D"    ! MOVES CURSOR TO LEFT
50   PRINT USING "#,K";"Dear __________"&RPT$(A$,10)
60   INPUT "Name?",Name$
70   PRINT Name$&",      "
80   PRINT USING "#,K";"Your subscription to __________"&RPT$(A$,10)
90   INPUT "Magazine?",Magazine$
100  PRINT Magazine$&" is ";LIN(1);"about to expire."
110  PRINT "Please renew it as soon as possible"
120  PRINT "so you don't miss any important news."
130  END
```

Using Tabs

The following escape code sequences are used to set and clear tabs –

ESC 1	Sets a tab at the column of the cursor
ESC 2	Clears a tab at the column of the cursor
ESC 3	Clears all tabs

The cursor can be moved to the next tab setting using ESC I or the control code HT (horizontal tab) which can also be accessed using CHR$(9). If no tabs are set a TAB moves the cursor to the beginning of the next line. Tabs remain set until they are cleared with ESC E, reset CLEAR or ESC 2 or ESC 3.

A

Clearing, Inserting and Deleting Lines

The following escape code sequences can be used in editing lines –

⁴⁄₁₆J	Clears the screen from the cursor position (remainder of the line and all lines following)
⁴⁄₁₆K	Clears the remainder of the line from the cursor position
⁴⁄₁₆L	Inserts a blank line before the cursor line. Cursor remains on same line of CRT and all following lines move down
⁴⁄₁₆M	Deletes the cursor line and closes up the gap. Cursor remains on same line of CRT, and all following lines roll up

These escape code sequences are very useful for text processing applications.

Inserting and Deleting Characters

The following escape code sequences can be used for inserting and deleting characters –

⁴⁄₁₆P	Deletes the character at the cursor position
⁴⁄₁₆Q	Turns on the insert character mode. Characters can be inserted to the left of the cursor. Insert mode remains in effect until it is cleared with ⁴⁄₁₆E, reset, CLEAR or ⁴⁄₁₆R.
⁴⁄₁₆R	Turns off the insert character mode

Example

```
  10   PRINT PAGE
  20   PRINT "THIS IS THE OLD TEXT"
  30   WAIT 2000
• 40   PRINT CHR$(27)&"&a1r12C";RPT$(CHR$(27)&"P",3);
• 50   PRINT CHR$(27)&"QNEW AND IMPROVED";CHR$(27)&"R"
  60   END
```

Rolling the Display

The display in the printout area of the CRT can be rolled using the following escape code sequences –

⁴⁄₁₆S	Rolls the printout up one line (like ↑)
⁴⁄₁₆T	Rolls the printout down one line (like ↓)
⁴⁄₁₆U	Rolls the printout area up 20 lines (next page)
⁴⁄₁₆V	Rolls the printout area down 20 lines (previous page)

When using the escape code sequence with S and T to roll the printout, the cursor stays in the same line of the CRT. Using any of the roll keys moves the cursor also. When using the escape code sequence with U and V, the cursor is positioned to the upper left hand corner of the CRT. The printout can only be rolled as far as the lines in memory; it can't be rolled past the existing lines to unused lines. You can't roll all existing lines off the screen.

These escape code sequences are useful for accessing a line that is not currently displayed, then moving the cursor in that line.

A

Selective Scrolling (Memory Lock)

Through use of an escape code sequence, it is possible to "freeze" a selected number of the upper lines of the CRT in place. The roll keys and escape code sequences used for rolling the display then have no effect on these lines. This is useful for keeping selected lines, such as the heading of a table, on the CRT while other lines are rolled up or down. This can be done with the following escape code sequence –

Ecl (lowercase L)	Freezes all lines which are above the cursor line.

The remaining bottom lines can scroll up or down without moving the frozen lines. However, absolute row addressing is disabled when memory lock is on. Output of a formfeed character won't clear the frozen lines. EcH positions the cursor to the first unfrozen line.

When memory lock is on, the cursor can't be positioned using R to address a row of memory. Y must be used to address a line of the printout area. When the printout is rolled using the U and V escape code sequences, the cursor is positioned to the first unlocked line.

The frozen lines remain frozen until cleared with CLEAR, EcE, or by using the following escape code sequence –

Ecm	Unfreezes the lines which were frozen previously.

Example

```
10   PRINT "X","SIN X","COS X","TAN X"
20   PRINT "_","_____","_____","_____"
30   PRINT CHR$(27)&"l"
40   FOR X=0 TO 2*PI STEP .1
50      PRINT X,SIN(X),COS(X),TAN(X)
60   NEXT X
70   END
```

Highlights, Color and Screen Addressing (Enhanced Graphics ROM)

When screen addressing is used to re-access a previously printed line, the previously printed highlight/color is preserved until a new highlight/color is specified. If a line is re-accessed, but to the right of previously printed characters, then the currently active highlight/color is used.

```
10   PRINT PAGE;
20   PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ" !Print this line at top of screen
30   PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ" !Print identical line under it
40   PRINT "EcH;"                       !Return to top of screen; linefeed
50   PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ" !Print this line over second line
60   PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ" !Print identical line under it
70   END
```

Program results:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

9845B Graphics ROM

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

9845C Graphics ROM

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Enhanced Graphics ROM

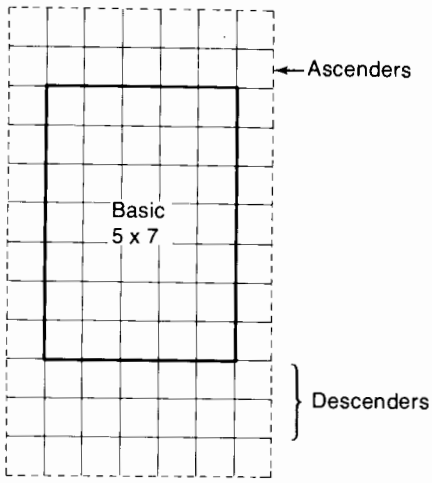
A

The Internal Printer

The internal thermal printer has additional capabilities, differing from some of those on the CRT, which allow you to generate new characters, replace any character with a string, set and clear tabs, alter margins, plot in plotting mode and print 150% size characters. These capabilities are all accessed by using an escape code (CHR$(27)) and parameters. Keep in mind that the escape codes remain in effect until they are deactivated. This can be done with ESCE or reset. Some of the following features also have an escape code sequence which specifically clears it.

Structure

The internal printer prints up to 80 printable characters per line. Each character is formed from a 5 x 7 dot matrix contained within a 7 x 12 dot matrix.



The two rows above the 5 x 7 dot matrix are used for ascenders such as an umlaut. The two descender rows are used for "legs" of lower case letters like p and y. The last row is used for underline.

Rows Per Line

The number of rows of dots per line (from baseline to baseline) of printing can be altered from the normal 12. Vertical spacing is set by outputting –

ESC&ldd$

l is a lower case "L". dd specifies one or more octal digits representing the number of rows of dots. Each row of dots represents 1/77 of an inch of vertical spacing.

The range of the digits is 0 through 126 (176 octal). Default is twelve.

Example

```
10    !  This example makes the printer double space
         all lines printed to it
20    !  After running the example, try listing the
         program;  it will be double spaced
30    PRINTER IS 0
40    PRINT USING "#,K";CHR$(27)&"&124S"
50       FOR I=1 TO 5
60       PRINT "I EQUALS:";I
70       NEXT I
80    END
```

Margins

When perforated paper is used, the top margin is normally set to approximately ½″ below the perforation. This amount can be altered by outputting –

Ec&lddT

l is a lowercase "L". dd are octal digits specifying the number of 1/77″ rows below the perforation as being the top margin.

The range of the digits is 0 through 127 (177 octal). 0 causes the perforations to be ignored; a line could conceivably be printed on top of the perforation on 2 pages. Default is 36.

The bottom margin has a width in the range .79 to 1.19 centimetres.

Example

```
10    !   This program causes the perforations on the
          paper to be ignored.  This could cause one
20    !   line to be printed on top of the perforation
          on two pages
30    PRINTER IS 0
40    PRINT USING "#,K";CHR$(27)&"&10T"
50    END
```

Setting Tabs

Horizontal tabs can be set by outputting –

Ec1

A tab is set at the current character position. It is used by outputting a TAB code, CHR$(9).

If no tabs are set or no more remain in the current line, a CR-LF is executed by the printer when a TAB code is received. Two horizontal tab anomalies are associated with string replacement; see the section on String Replacement.

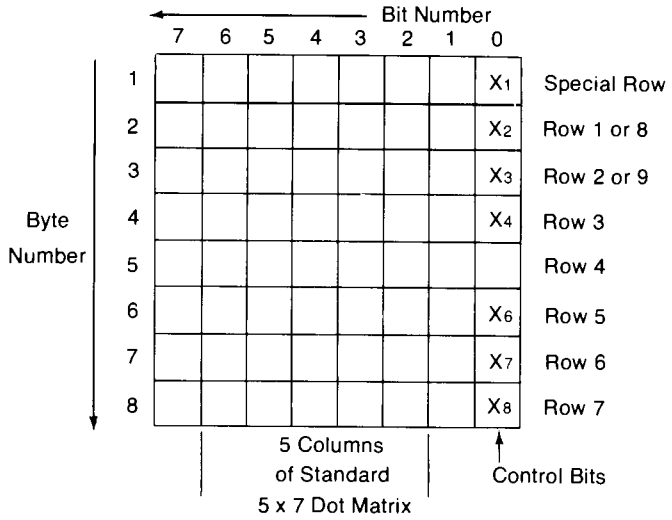
All horizontal tabs are cleared with –

Ec3

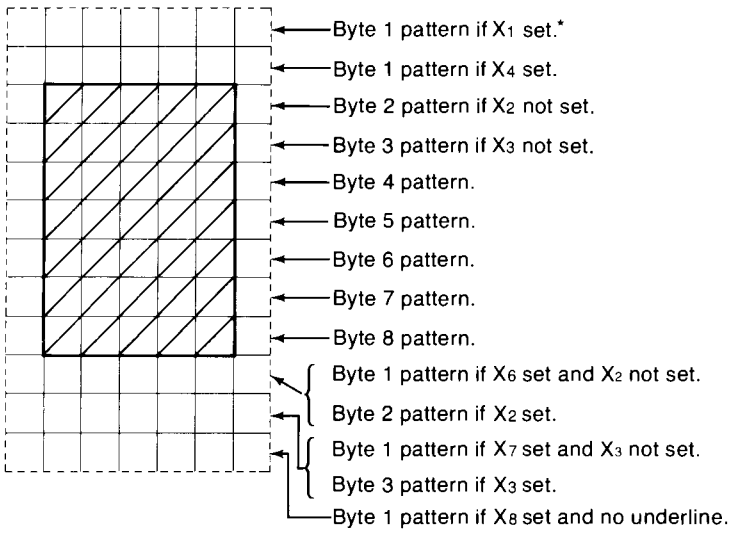
New Characters

Up to nine new characters which are not part of the standard or alternate character sets can be defined to replace another character.

A new character is defined by specifying a character to represent it and specifying up to eight 8-bit byte patterns to define the character. Bits one through seven of each byte are printed; bit 0 is used as a control bit and is specified by a subscripted X.



These eight bytes are used to define up to twelve rows of dots for the new character. Byte one defines a special row which can be printed in any of five row positions. Bytes two and three define the first two rows of a standard 5 x 7 dot matrix; either or both can be moved to define the two rows below the 5 x 7. Control bits are set when they equal one. This is used to determine where bytes one, two and three are printed.



* If the number of rows per line is less than twelve, byte 1 pattern is printed in the first row above the standard 5 x 7 when X1 is set.

The syntax used to define a new character is –

ESC&ndd c dd p dd q dd r dd s dd t dd u dd v dd W

The n specifies that this is a new character definition.

dd specifies one, two or three octal digits; the letter following them defines their purpose.

The digits preceding the c are the octal digits representing the character which is to be replaced by the character being defined.

The bytes being defined by their octal equivalent are represented as shown –

Letter	Byte
p	1
q	2
r	3
s	4
t	5
u	6
v	7
w	8

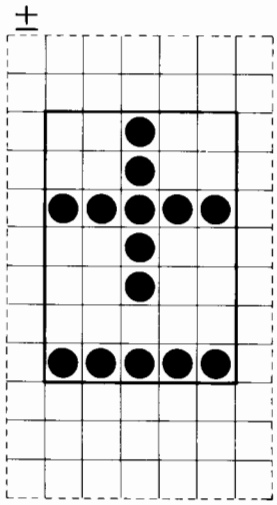
The last letter in the sequence **must** be a capital letter. Thus, if bytes 7 and 8 are not specified, the u must be capitalized.

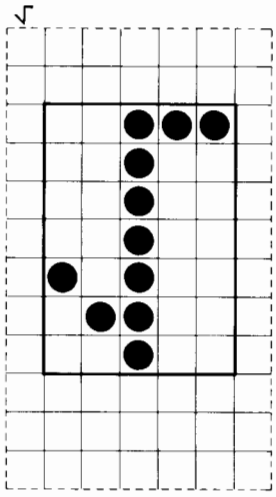
It is not necessary to define all of the bytes. Thus, byte four can be left undefined by omitting the s and preceding digits. However, if these bytes were previously defined in another new character, the old byte definitions remain. ESCE clears old definitions.

Example

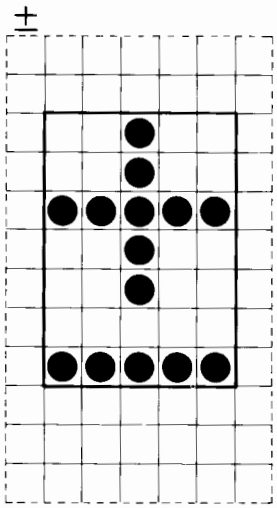
Here is an example which illustrates new character definition. Say that you wanted to print the formula for solving quadratic equations, Y= −B±√ (B∧2−4AC)/2A. The ± and √ symbols can be generated quite easily.

The first step is to block out the characters being defined.

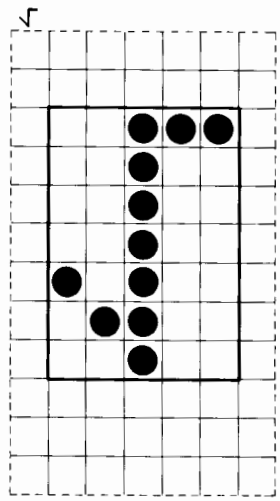




The second step is to determine the 7-bit pattern and octal representation. The control bit is shown in parentheses.



Bit Pattern	Octal
0001000(0)	20
0001000(0)	20
0111110(0)	174
0001000(0)	20
0001000(0)	20
—	—
0111110(0)	174



Bit Pattern	Octal
0001110(0)	34
0001000(0)	20
0001000(0)	20
0001000(0)	20
0101000(0)	120
0011000(0)	60
0001000(0)	20

Next, determine characters which the defined characters will replace.

- replace % with √
- replace ′ with ±

This program prints the formula Y= −B± √ (B∧2−4AC)/2A.

```
  10   DIM A$[80],B$[80]
  20   E$=CHR$(27)          ! Escape code
• 30   A$=E$&"&n45c34q20r20s20t120u60v20W"
• 40   B$=E$&"&n47c20q20r174s20t20u174W"
  50                        ! A$ & B$ define square root
                              and plus-minus signs
  60   PRINTER IS 0
  70   PRINT A$,B$          ! Nothing is printed until line
                              90, which utilizes the new
  80                        ! characters, is executed
• 90   PRINT "Y=-B′%(B^2-4AC)/2A"
  100  PRINT E$&"E"         ! Resets printer
  110  END
```

```
Y=-B±√(B^2-4AC)/2A
```

---

77 bytes of memory are reserved for new character definition and string replacement (discussed next). Each new character uses 8 bytes of memory. Thus, a maximum of nine new characters can be defined at one time; an additional definition replaces the last character defined.

String Replacement

Any character can be replaced by a string. The syntax is –

```
[c&odd:ddLtext
```

The digits preceding the c are the octal equivalent of the character which is being replaced. The digits preceding the L are the octal equivalent of the length in characters of the replacing string.

If a character is both redefined and replaced, the replace takes precedence over the definition. Of course, a character can be in the string it is replaced by and if it is redefined, is printed redefined.

Example

```
10   E$=CHR$(27)                  ! Escape code
20   PRINTER IS 0
30   PRINT E$&"E"                 ! Clear all control codes
40   PRINT LIN(5),SPA(25);"CHARACTER REPLACEMENT",LIN(2)
50   PRINT SPA(25);"JACK + JILL"
60   PRINT E$&"&o53c3Land"        ! Replace '+' (octal 53) with
                                    'and' (length 3)
70   PRINT LIN(1),SPA(25);"JACK + JILL",LIN(5)
80   PRINT E$&"E"                 ! Clear all control codes
90   END
```

```
CHARACTER REPLACEMENT

JACK + JILL

JACK and JILL
```

Remember, 77 bytes total are available for new characters and string replacement. Also, the maximum number of string replacements would be 25 single-character replacements. Each string replacement requires 2 bytes plus one for each character in the string. Thus, a maximum string length is 75 characters for replacement. Any additional attempts at replacement are ignored.

Two anomalies are associated with string replacement and horizontal tab. If HT is contained in a replace string and no tabs are set, the printer will keep searching for a tab, feeding paper as it goes. Reset (CONTROL-STOP) must be used to abort this.

The second anomaly arises when an HT immediately follows a replace string definition. It is ignored if it is the **next** character specified to be output after the replace string definition.

150% Size Characters

Any character can be printed 150% of the normal character size. The syntax used to accomplish this is –

```
Ec&k1S
```

To get back to normal size characters, use this sequence –

```
Ec&k0S
```

Example

```
  10   PRINTER IS 0
  20   E$=CHR$(27)
• 30   PRINT "XXXX"&E$&"&k1SXXXX"   ! Normal then large
• 40   PRINT E$&"&k1SXXXX"&E$&"&k0SXXXX"
  50   END
```

```
XXXXXXXX
XXXXXXXX
```

Underlining

Any characters can be underlined by using the following syntax –

Ec&d underline indicator

An underline indicator is a character used in the above sequence both to begin the underlining and to turn it off. The following table shows the various underline indicators and their function.

	Characters
Underline	D, E, F, G, L, M, N, O
End Underline	@, A, B, C, H, I, J, K

Example

```
  10   PRINTER IS 0
  20   E$=CHR$(27)          ! Escape code
• 30   PRINT E$&"&dDFINAL"&E$&"&dA NOTICE"
  40   END
```

```
FINAL NOTICE
```

Underlining and 150% characters can be used at the same time. Underlining can also be accessed with CHR$(132) and turned off with CHR$(128).

Plotting Mode

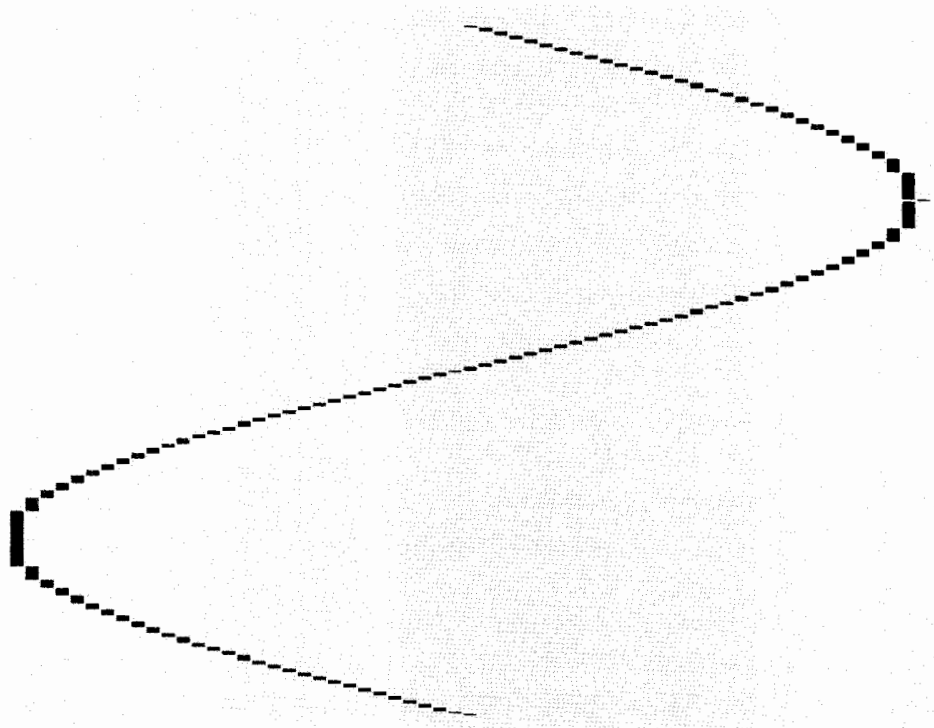
In plotting mode, the next 70 8-bit byte patterns define the dot pattern for the row. To access the plotting mode, output –

Ec?

Example

Here is an example of using the plotting mode to plot the sine function.

```
10    !   This program illustrates plotting mode
20    !   NOTE:  It has to do a lot of number-
                 crunching and takes a while to run
30    OPTION BASE 1
40    PRINTER IS 0
50    DEG
60    E$=CHR$(27)              ! Escape code
70    PRINT E$&"E"             ! Clear printer
80    DIM A$(360)[72]
90    FOR I=1 TO 360
100      A$(I)[1,2]=E$&"?" ! Enter plotting mode
110      A$(I)[3,72]=RPT$(CHR$(0),70)   ! Clears string
120   NEXT I
130   FOR I=1 TO 360
140      A$(I)[33+INT(30*SIN(I)),33+INT(30*SIN(I))]=CHR$(127)
150                           ! Use bit pattern for CHR$(127)
                                (01111111) & make sine wave
160   NEXT I
170   PRINT USING "#,K";A$(*)
180   END
```



A

Summary of Escape Codes

Escape Code Sequence	Action	For CRT	For Internal Printer
ESC A	Moves cursor up one row	√	
ESC B	Moves cursor down one row	√	
ESC C	Moves cursor right one column	√	
ESC D	Moves cursor left one column	√	
ESC E	Resets the printer and CRT – clears control features	√	√
ESC F	Moves cursor to row after last row of CRT memory, first column	√	
ESC G	Moves cursor to first column, current row	√	
ESC H	Moves cursor to first row of CRT memory, first column	√	
ESC I	Moves cursor to next tab setting	√	
ESC J	Clears screen from cursor (rest of line and all lines following)	√	
ESC K	Clears line from cursor position	√	
ESC L	Inserts a blank line before cursor line	√	
ESC M	Deletes cursor line and closes up gap	√	
ESC P	Deletes character at cursor position	√	
ESC Q	Turns on insert character mode; inserts to left of cursor	√	
ESC R	Turns off insert character mode	√	
ESC S	Rolls printout up one line (like [+])	√	
ESC T	Rolls printout down one line (like [+])	√	
ESC U	Rolls printout up 20 lines (next page)	√	
ESC V	Rolls printout down 20 lines (previous page)	√	
ESC Y	Disables control codes and allows them to be viewed	√	√
ESC Z	Reactivates control codes	√	√
ESC l (lowercase L)	Freezes all lines above cursor line	√	
ESC m	Unfreezes the lines which were frozen previously	√	
ESC 1	Sets a tab at column of the cursor	√	√
ESC 2	Clears a tab at column of the cursor	√	

ESC 3	Clears all tabs	√	√
ESC &a	Addresses the cursor	√	
ESC &d	Accesses CRT special features	√	√
ESC &k	Outputs 150% size characters		√
ESC &l (lowercase L)	Sets dots per line or top margin		√
ESC &n	Defines a new character		√
ESC &o	Specifies a string replacement		√

Examples

The first example listed is used to move blocks of text. The second example should be run to see how it lets you manipulate a table.

Example 1

```
10   !   This program uses CRT addressing to move blocks of text
20   !   This section outputs the text     ************************
30   PRINTER IS 16                        ! sets CRT as printer
40   PRINT PAGE;                          ! clears the CRT
50   PRINT "3.  This is the third paragraph."
60   PRINT "    It should be the last one"
70   PRINT "    in the group."
80   PRINT "2.  This is the second para-"
90   PRINT "    graph.  It should be the"
100  PRINT "    second one."
110  PRINT "1.  This is the first paragraph"
120  PRINT "    in the group.  It should"
130  PRINT "    be first."
140  PRINT
150  DISP "PRESS CONTINUE TO START ORDERING"
160  PAUSE
170  !  This section puts paragraphs in right order  ************
180  Ec$=CHR$(27)                         ! escape code
190  E$=CHR$(27)&"&a"                     ! escape code & '&a'
200 I1:   IMAGE #,K                       ! IMAGE for PRINT USING
210  PRINT USING I1;E$&"3Y"               ! move cursor to 1st line
                                            of paragraph 2
220  PRINT USING I1;Ec$&"l"               ! turn on memory lock
230  PRINT USING I1;RPT$(Ec$&"S",6)       ! roll remaining lines
                                            off screen
240  PRINT USING I1;Ec$&"m"               ! turn off memory lock
250  PRINT USING I1;Ec$&"H"               ! move cursor to 1st row,
                                            1st column
260  DISP "PRESS CONTINUE TO GO ON"
270  PAUSE
280  DISP                                 ! clear display
290  PRINT USING I1;E$&"3R"               ! move cursor to paragraph 1
300  PRINT USING I1;Ec$&"l"               ! turn on memory lock
310  PRINT USING I1;RPT$(Ec$&"S",3)       ! roll til cursor is in
                                            paragraph 3
320  PRINT USING I1;Ec$&"m"               ! turn off memory lock
330  PRINT USING I1;Ec$&"H"               ! move cursor to 1st row,
                                            1st column
340  END
```

A

Example 2

```
10   !   This program uses various CRT addressing operations
         to create and manipulate a tabel.  Each line of the
20   !   table is numbered to reflect the line number of CRT
         memory
30   STANDARD                          ! for number output
40   PRINTER IS 16                     ! CRT is printer
50   OPTION BASE 1
60   DIM Expenses(17,4),Accounts$(17) ! arrays for types and $'s
70   DATA  25,40,22.38,75,100,205.75,82,172.30,20,7.50,75.36,49
80   DATA  2000,827,40,1537,7,0,50.75,41,5000,4700,5130,4900,50,20
90   DATA  75,125,400,700,0,95,276,347,172.50,99.30,52.13,41.26
100  DATA  13,25.52,20,40,0,55,10,0,15,12,62,17.32,36.21,72.47
110  DATA  30.75,69.85,22,50,237,845,99,49,5,15,75,0,40,99,1275,83
120  DATA  Travel,Motels,Off. supplies,Machines,Entertainment,Pay
130  DATA  Overtime,Desks,Printing,Postage,Accts Payable,Donations
140  DATA  Miscellaneous,Rental cars,Advertising,Fees,Tooling
150  MAT READ Expenses,Accounts$       ! get values into arrays
160  Ec$=CHR$(27)                      ! escape code
170  E$=CHR$(27)&"&a"                  ! escape code & '&a'
180  Ts$=Ec$&"1"                       ! set tab at cursor position
190  T$=CHR$(9)                        ! horizontal tab
200  Del$=Ec$&"M"                      ! delete cursor line
210  Insl$=Ec$&"L"                     ! insert blank line above
                                         cursor line
220  Rol$=Ec$&"V"                      ! roll printout down
230 I1:   IMAGE #,K                    ! image for PRINT USING
240 Heading:  ! THIS SECTION OUTPUTS THE HEADING  ***************
250  PRINT " 0  ";RPT$("*",76),LIN(1)," 1",LIN(1)," 2"
260  PRINT " 3     Expense";TAB(25);"January";TAB(40);"February";
270  PRINT TAB(55);"March";TAB(70);"April"
280  PRINT " 4",LIN(1)," 5  ";RPT$("*",76)
290  L$=CHR$(124)                      ! vertical bar for spacing
300  PRINT " 6";TAB(20);L$;TAB(35);L$;TAB(50);L$;TAB(65);
310  PRINT L$;TAB(80);L$
320  PRINT E$&"7R"&Ec$&"1"             ! put memory lock on heading
330 Table:  ! THIS SECTION OUTPUTS THE TABLE  ********************
340  PRINT USING I1;E$&"24C"&Ts$       !
set tab at cursor position
350  PRINT USING I1;E$&"39C"&Ts$
360  PRINT USING I1;E$&"54C"&Ts$
370  PRINT USING I1;E$&"69C"&Ts$
380  PRINT USING I1;E$&"7y0C"          ! reposition cursor to row 7
390  Line=7
400  FOR I=1 TO 17                     ! these loops print table
410     PRINT USING "#,DD,K";Line,"    "&Accounts$(I)
420     FOR J=1 TO 4                   ! output figures
430        PRINT T$;Expenses(I,J);     ! T$ tabs to next setting
440     NEXT J
450     PRINT                          ! go to next line
460     Line=Line+1                    ! output line number of CRT
470  NEXT I
480 Change:  ! THIS SECTION ALLOWS NUMBERS TO BE CHANGED  ********
490  INPUT "Are there any figures you wish to change(Y OR N)?",A$
500  IF A$="N" THEN Reposition         ! branch to next section
510  INPUT "Enter row number of expense you want to change",Row
520  R$=VAL$(Row)
```

Computer Museum


```
530  INPUT "Which month do you want to change(J,F,M or A)?",Month$
540  IF Month$="J" THEN Rep=1          ! Rep is for tabbing purposes
550  IF Month$="F" THEN Rep=2
560  IF Month$="M" THEN Rep=3
570  IF Month$="A" THEN Rep=4
580  INPUT "Enter the new figure(<=9 digits)",Expenses(Row-6,Rep)
590  X=Expenses(Row-6,Rep)
600  PRINT Rol$                        ! roll printout down
610  IF Row>19 THEN Roll
620  PRINT USING I1;E$&R$&"y0C"        ! move cursor to right line
630  PRINT RPT$(T$,Rep);X;SPA(5)       ! tab to proper column,
                                         print number
640  INPUT "Are there more figures you want to change(Y or N)?",A$
650  IF A$="N" THEN Reposition         ! go to next section
660  GOTO 510                          ! back to start of this section
670  STOP
680 Roll:   !   THIS SECTION COMPENSATES FOR ROWS > 19  *********
690  PRINT USING I1;RPT$(Ec$&"S",Row-19)   ! position row in row 19
700  PRINT USING I1;E$&"19Y"           ! position cursor in row 19
710  PRINT RPT$(T$,Rep);X;SPA(5)       ! tab to proper column,
                                         print number
720  GOTO 640
730  STOP
740 Reposition:  ! THIS SECTION LETS A LINE BE MOVED  ************
750  INPUT "Which line do you want to move?",L1
760  INPUT "Which line do you want it to go above?",L2
770  L1$=VAL$(L1)
780  L2$=VAL$(L2)
790  PRINT USING I1;Rol$               ! roll printout down
800  IF (L1>19) OR (L2>19) THEN Roll2
810  IF L1<L2 THEN L2$=VAL$(L2-1)      ! make up for deletion of L1
820  PRINT USING I1;E$&L1$&"Y"         ! move cursor to line moving
830  PRINT USING I1;Del$               ! delete line being moved
840  PRINT USING I1;E$&L2$&"Y"         ! move cursor to L2
850  PRINT USING I1;Ins1$              ! insert blank line above L2
860  PRINT USING "#, DD,K";L1,"   "&Accounts$(L1-6) !line# & acc't
870  PRINT RPT$(" ",13-LEN(Accounts$(L1-6)));  ! for shorter names
880  FOR I=1 TO 4                      ! reprint the deleted line
890     PRINT T$;Expenses(L1-6,I);
900  NEXT I
910  DISP "PRESS CONTINUE TO RENUMBER THE LINES"
920  PAUSE
930  PRINT USING I1;Rol$               ! roll down
940  PRINT USING I1;E$&"7y0C"          ! position cursor to line 7
950  FOR I=7 TO 23                     ! renumber all lines from 7
960     PRINT USING "DD";I
970  NEXT I
980  DISP                              ! clears display
990  STOP                              ! PROGRAM STOPS HERE ******
1000 Roll2:   ! EITHER OR BOTH LINES > THAN 19   ****************
1010 IF (L1>19) AND (L2<=19) THEN R1
1020 IF (L1<=19) AND (L2>19) THEN R2
1030                                   ! both L1 and L2 > 19
1040 PRINT USING I1;RPT$(Ec$&"S",4)    ! roll last line to line 19
1050 L1$=VAL$(L1-4)                    ! compensates for rolling
1060 L2$=VAL$(L2-4)
1070 IF L1<L2 THEN L2$=VAL$(L2-5)      ! make up for deleted line
1080 GOTO 820
```

```
1090 R1:    ! L1>19 AND L2<=19     ********************************
1100 PRINT RPT$(Ec$&"S",L1-19)          ! roll line being moved to 19
1110 PRINT USING I1;E$&"19y0C"          ! move cursor to that line
1120 PRINT USING I1;Del$                ! delete line being moved
1130 PRINT USING I1;Rol$                ! roll printout down
1140 GOTO 840
1150 R2:    ! L1<=19 AND L2>19    ********************************
1160 PRINT USING I1;E$&L1$&"Y"          ! put cursor in line moving
1170 PRINT USING I1;Del$                ! delete line being moved
1180 PRINT USING I1;RPT$(Ec$&"S",3)     ! roll last line to line 19
1190 L2$=VAL$(L2-4)                     ! compensates for rolling
1200 GOTO 840
1210 END
```

Appendix B
Programming Exercises

Exercises

This appendix contains several exercises to let you practice creating flowcharts, program outlines and programs. The solutions are at the end of the appendix.

Exercise 1

Construct a flowchart to output all odd numbers between 35 and 50.

Exercise 2

Construct a program outline to calculate and print a compound interest table for $1000 of initial principal, 6% interest and 100 periods. A compound interest table has two columns of numbers. The first is the compounding period; the second is the new principal amount after compounding the interest for that period, which is obtained by multiplying the previous principal by the interest rate, then adding that product to the previous principal. Repeat this procedure for as many periods as you want.

Exercise 3

Modify the previous program outline to allow the user to input the initial principal, interest and number of periods and to repeat the procedure if he wants.

Exercise 4

Write a program which computes the straight-line distance between two points. Then input the x,y-coordinates of both points. (The distance formula is $\sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$.)

Exercise 5

Write, store and execute a program which prints the numbers 35 to 50 using the READ and DATA statements.

Exercise 6

Write, store and execute a program which has three sets of numbers (three numbers in each set). Corresponding numbers in the first two sets are added together, and the resulting sum is multiplied by the corresponding number in the third set. Read in the sets using READ statements. The numbers are:

1st set = [4,17,−3] 2nd set = [8,4,−2] 3rd set = [12.1,7.33,5]

B

Exercise 7

Write, store and execute a program which calculates withholding tax. Withholding tax depends upon both the salary amount and the number of exemptions. Taxable income is the salary less $14.40 for each exemption. The actual amount withheld is $6.60 on the first $105, plus 18% of the taxable income above $105. Display both the tax withheld and net salary (salary minus withholdings).

Exercise 8

A mathematician, Karl Friedrich Gauss, developed a theorem that said the sum of the numbers from 1 to some other number, say n, could be represented by the equation $S = n(n+1)/2$. It holds for the number 3 certainly, because $1+2+3=6$ and $3(3+1)/2 = 6$, but does it hold for all numbers? Write and run a program which will count (1,2,3,4,...), add up the numbers as it counts, and compare the sums to the results predicted for the formula for each number. Display the results on the CRT. Are you in an infinite loop? Does the formula hold as far as you let it run?

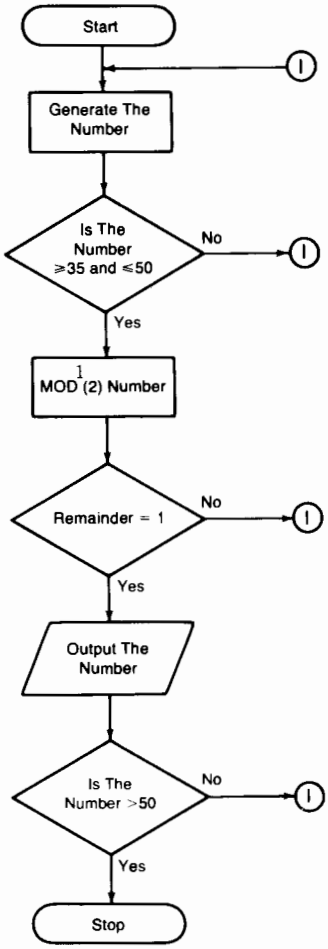
Exercise 9

Write and run a program which computes the standard deviation of a sequence of numbers. The user should be able to enter the sequence. The formula for standard deviation is

$$\sigma = \sqrt{\frac{\Sigma x^2 - \frac{(\Sigma x)^2}{n}}{n-1}}$$

Answers to Exercises

Computer Museum



Exercise 2

1. Set the principal, interest rate and number of periods.
2. Print the table heading.
3. Generate and print the period number.
4. Compute and print the principal ((Principal*Rate) + Principal).
5. If there is another compounding period, go to step 3.

Exercise 3

Add this step to the program outline for Exercise 2:

6. If another table is desired, go to step 1.

[1] MOD is an operator.

Exercise 4

```
10   INPUT "X1 and Y1",X1,Y1
20   INPUT "X2 and Y2",X2,Y2
30   Straight_line=SQR((X1-X2)^2+(Y1-Y2)^2)
40   PRINT Straight_line
50   END
```

RUN (example inputs where X1=4, Y1=5, X2=7, Y2=8)

B

```
4.24264068711
```

Exercise 5

```
10   DATA 35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50
20   READ A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P
30   PRINT A;B;C;D;E;F;G;H;I;J;K;L;M;N;O;P
40   END
```

RUN

```
35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
```

Exercise 6

```
10   DATA 4,17,-3
20   DATA 8,4,-2
30   DATA 12.1,7.33,5
40   READ A,B,C,D,E,F,G,H,I
50   X=(A+D)*G
60   Y=(B+E)*H
70   Z=(C+F)*I
80   PRINT X,Y,Z
90   END
```

RUN

```
145.2              153.93             -25
```

The program can be rewritten from line 50 as:

```
50   PRINT (A+D)*G,(B+E)*H,(C+F)*I
60   END
```

Exercise 7

```
10   FIXED 2
20   INPUT "Salary",Salary,"Exemptions",Exemptions
30   Taxbase=Salary-Exemptions*14.4
40   Overtax=Taxbase-105
50   Tax=6.6+Overtax*.18
60   PRINT "Tax Withheld $";Tax,"Net Salary $";Salary-Tax
70   END
```

(RUN) **(example inputs)**

```
Tax Withheld $ 29.13                    Net Salary $ 229.83
```

Exercise 8

```
10   A=N=0
20   N=N+1
30   A=N+A
40   DISP N;
50   IF N*(N+1)/2=A THEN 90
60   DISP "Invalid Theorem"
70   STOP
80   DISP A,"Valid Theorem"
90   GOTO 30
100  END
```

With the System 45 Desktop Computer, you can eliminate line 10 because variables are **preinitialized** to zero. However, initializing variables to zero is a wise procedure to follow in general programming.

Exercise 9

```
10   X=N=X_bar=0
20   INPUT "Next number in the sequence",Y
30   PRINT "Previously entered";Y
40   X_bar=X_bar+Y
50   X=X+Y^2
60   N=N+1
70   INPUT "Enter 1 for more number in the sequence; enter 0 if done",A
80   IF A=1 THEN 20
90   Sigma=SQR((X-X_bar^2/N)/(N-1))
100  PRINT "Standard deviation";Sigma
110  END
```

(RUN)

```
Previously entered 45
Previously entered 89
Standard deviation 31.1126983722
```

Computer Museum

Reference Tables

Glossary

Absolute plotting – Plotting to a coordinate which has its X and Y values specified in the current user units.

Angle – The vector which a line or label is drawn, represented in degrees, radians, or grads, counter-clockwise from the horizontal.

Anisotropic – The X and Y units are not displayed as equal to each other.

Array identifier – An array name followed by (*) used to access all elements in an array collectively.

Axes – plural of axis.

Axis – A line drawn within the Cartesian coordinate system along either the horizontal (X) direction or the vertical (Y) direction.

Byte – A group of 8 binary digits (bits) operated upon as a unit.

Calling program – When a subprogram is being executed, the program segment (main program or subprogram) which called the subprogram is known as the calling program. Control returns to the calling program when the subprogram is completed.

Character – A letter, number, symbol, or ASCII control code; any artitrary 8-bit byte defined by the CHR$ function.

Clipping area – The area which restricts the pen movement whenever lines are drawn in UDUs.

Command – An instruction to the computer which is executed from the keyboard. Commands are executed immediately, do not have line numbers and can't be used in a program. They are used to manipulate programs and for utility purposes, such as listing key definitions.

Constant – A fixed numeric value within the range of the computer; for example, 29.5 or 2E12.

Controller address – An integer from 0 through 7 which specifies the address of a hard disc controller. 0 is the default address.

Current environment – The program segment which is being executed.

Current units – The mode of X,Y reference which is in effect; may be GDUs, UDUs or Metric (mm).

Cursor – The device which is used to obtain digitizing information.

Data base – A set of data which is accessible by the computer and upon which a program may perform operations.

Defined record – The smallest unit of storage on a mass storage medium which is directly addressable. A defined record is established using the `CREATE` statement and can be specified as having any number of bytes in the range 4 through 32 767 (rounded up to an even number).

Digitizing – The process of obtaining an X,Y coordinate pair based on the location of the cursor.

Display line – Line 22 of the CRT is used to display output generated by `DISP`, and any `INPUT` prompt or question mark.

Edit key mode – When a Special Function Key is being defined as a typing aid. See `EDIT KEY`.

Edit line mode – When the program in memory is being edited. See `EDIT LINE`.

File name – A one to six character string expression with the exception of a colon, quote mark, ASCII NULL (CHR$(0)), or CHR$(255). Blanks are ignored.

File number – The number assigned to a mass storage data file by an `ASSIGN` statement. Its range is 1 through 10.

File pointer – The current position within a file where data is about to be read or written.

File specifier – A string expression of the form: file name [mass storage unit specifier].

Files – The basic unit into which programs and data are stored. Storage of all files are "file-by-name" oriented; that is, each file must be assigned a unique name.

Formal parameter – Used to define subprogram variables and can be non-subscripted variables, array identifiers or files specified by #file number. A type word can come before parameters to specify numeric type. Parameters must be separated by commas; the parameter list must be enclosed in parentheses.

GDUs – Graphic Display Units. An X,Y reference system which at default defines the CRT to extend from X minimum = 0 to X maximum = 123.127753304, Y minimum = 0 to Y maximum = 100.

Handshake – A signal exchange between the computer and external device to communicate data ready and data accepted information.

Hard clip – The physical limits of the plotting device, beyond which no line can be drawn.

HPGL – The low-level instruction set used with HP input and output devices.

HP-IB device address – A numeric expression which specifies the HP-IB address that is set on a device. Its range is 0 through 30.

Input – A data transfer from an external device to the computer.

Interleave factor – Defines the number of revolutions per track to be made for a complete data transfer on a 9885 or 9895 Disc. It is specified in an `INITIALIZE` statement.

Isotropic – The X and Y axes units are displayed as equal to each other.

Keyboard entry area – Lines 23 and 24 of the CRT are accessible only through keyboard inputs. Every line that is typed in is displayed in this area. The first position in line 23 is known as the "home" position of the cursor. As the 148th character is keyed in, a beep indicates that only 12 more characters can be entered.

Label – A unique name given to a program line. It follows the line number and is followed by a colon.

Line identifier – A program line can be identified either by its line number (`GOTO 150`) or its label, if any, (`GOTO Routine`).

Line number – An integer from 1 through 32 766. In most cases, when a line number is specified, but is not in memory, the next highest line is accessed.

Live keyboard mode – Numeric computations and most statements and commands can be executed from the keyboard while a program is running. Program lines can be stored also. The running program is temporarily paused while a keyboard operation is executing.

Local variable – A variable in a subprogram that isn't declared in the formal parameter list or COM statement; it can't be accessed from any other program segment. Storage of local variables is temporary and returned to user Read/Write Memory upon return to the calling program.

Logical record – A collection of data items which are conceptually grouped together for mass storage operation. It is a user-level rather than a machine concept.

Main program – The central part of a program from which subprograms can be called is known as the main program. When you press RUN, you access the main program. The main program can't be called by a subprogram.

Mass storage unit specifier (msus) – Any string expression of the form –

:device type [select code[, controller address | 9885 unit code [, unit code]]]

The letters specifying the various mass storage device types are –

Device Type Code	Device
T	Internal tape cartridge
Y	7905M removeable disc
Z	7905M fixed disc
C	7906M/H removeable disc
D	7906M/H fixed disc
M	7910H fixed disc (98413B ROM only)
P	7920M/H removeable disc
X	7925M/H removeable disc
F	9885M/S flexible disc
H	9895A flexible disc
Q	7908 drive

Medium – The material on which data is actually being kept and stored (as distinct from the device, which does the actual reading and writing). Tape cartridges and disc packs are examples of "media".

Metric units – A unit of measure mode where everything is referenced in millimetres.

Mnemonic – An abbreviation or acronym that is easy to remember.

Module – In programming, a program segment which performs a specific, independent program task.

Computer Museum

Msus – The abbreviation for mass storage unit specifier.

Name – A capital letter followed by 0 through 14 lower case letters, digits or the underscore character. Names are used for variable names, labels, function names, and subprogram names.

Naming convention – A pattern or system for assigning names to variables or files so that some manner of consistency or predictability is maintained.

Numeric expression – A logical combination of variables, constants, operators, functions, including user-defined functions, grouped with parentheses if needed.

On-line – Capable of being accessed by the computer; usually means a device which is physically connected, functioning properly, and in communication with the mainframe.

Origin – The coordinate point at which a plotting operation begins.

Output – A data transfer from the computer to an external device.

Parity – A means of flagging data transmission errors by setting the eighth bit to produce an even or odd number of set bits in a data word.

Pass parameter – Used in calling a subprogram to pass values to the subprogram and can be variables, array identifiers, expressions or files specified by #file number; any variable can be enclosed in parenthesis causing it to be passed by value.

Pen – The device which is used to draw or plot lines, and to label characters.

Physical record – A 256-byte, fixed unit which is established when a mass storage medium is initialized. Every file starts at the beginning of a physical record; this is an important fact for optimum device use. Otherwise, you need not be concerned with physical records.

Pixel – Picture element – the smallest unit of resolution on the CRT.

Plotted point – The point which has been plotted or drawn to.

Plotting coordinates – The X,Y coordinate pair which specifies a plotting point.

Plotting space – The area within which plotting can occur.

Pointer – The method used to position the cursor, or to select a type of cursor.

Printout area – Lines 1 through 20 of the CRT are similar to a printing device. When the machine is switched on, this area is the standard system printer to which output from PRINT, PRINT USING, CAT and LIST is directed. It is also, at power-on, the print all printer when in the print all mode.

Priority – A number in the range 1 through 15 which determines whether or not and in what order interrupts are serviced. The priority of the interrupt must be higher than current system priority to be serviced.

Program segment – The main program and each subprogram are known as program segments. Every program segment is independent of every other program segment.

Protect code – Any valid string expression except one with a length of zero. Only the first six characters are recognized as the protect code, however.

Read Only Memory (ROM) – Permanent memory which can't be changed or erased. Option ROMs are used to expand the language and capabilities of the computer.

Read/Write Memory (RWM) – Used to store programs, data and related information. The information in Read/Write Memory can be changed and is lost when the computer is shut off.

Record I/O – Input/output operations concerned exclusively with the smallest addressable unit of storage (records).

Redim subscripts – Numeric expressions separated by commas and enclosed in parentheses.

Reflected plot – A plot produced by interchanging either the X minimum and X maximum coordinates, the Y minimum and Y maximum coordinates, or both X and Y coordinate pairs to change the plot.

Relative plotting – Plotting which specifies plotting from an origin rather than to a specific X,Y coordinate.

Scalar – A numeric expression used as a constant in mathematical operations.

Select code – An expression (rounded to an integer) in the range 0 through 16 which represents an interface address. The following select codes are reserved by the system and can't be set on an interface:

- 0 Internal thermal printer and keyboard
- 13 Graphics
- 14 Left tape drive
- 15 Right tape drive
- 16 CRT

Slant – The angle at which a character is drawn, represented in clockwise degrees, radians, or grads from the vertical.

Snapshot – Current state at a particular time.

Soft clip – The limits of the plotting device which restrict pen movement for lines drawn in UDU's.

Special Function Keys (SFK's) – These keys can be defined or redefined for use as typing aids for statements, variable names or other series of keystrokes which are used often. Many of them have pre-defined definitions. Any of the special function keys can also be defined to have program interrupt capability.

Stack – A portion of memory used to temporarily hold information for processing in a particular order.

Standard mass storage device – The device to which all mass storage operations are directed if no device is specified. It is the righthand tape cartridge at power on and can be changed using the `MASS STORAGE IS` statement.

Standard printer – The printer to which all `PRINT`, `PRINT USING`, `CAT` and `LIST` output is directed if no device is specified. At power on, it is the CRT. It can be changed using the `PRINTER IS` statement.

Statement – An instruction to the computer telling it what to do while a program is running. A statement can be preceded by a line number, stored and executed from a program. Most statements can also be executed from the keyboard without a line number.

String expression – As with numbers, you can manipulate strings, thus forming a string function. The different forms of a string expression are text within quotes, string variable name, substring, string concatenation operation, string function, and user-defined string function.

Subprogram – A set of statements, separate from and after the main program, that performs a task under the control of the calling program segment. SUB and CALL or DEF FN and FN are used to define and access a subprogram.

Subroutine – A set of statements, within a program segment, that performs a task. The GOSUB and RETURN statements control subroutines.

Subscript – An integer used to specify the range of an array dimension. A single subscript is used to specify the upper bound of a dimension; two subscripts separated by a colon are used to specify the upper and lower bounds of a dimension. A comma is used to separate the subscripts for each dimension.

System comments line – Line 25 of the CRT is reserved for error messages, mode indicators, and the run light: ▒ Results of keyboard operations, such as 3+5 EXECUTE or X EXECUTE, also appear in this line.

System design – The specification and implementation of a program or set of programs to accomplish a given purpose.

Text – Any combination of characters; for example "ABC". Text can be quoted (literal) or unquoted.

UDUs – User Defined Units. Defined by the program to whatever X and Y units of measure which are convenient.

Unit code – The address set on a hard disc drive; it can be an integer from 0 through 7. 0 is the default code. It is ignored for the 9885 and tape cartridge.

The 9885 unit code is the address set on a 9885 disc drive; it can be an integer from 0 through 3. 0 is the default code.

Variable – A name which is assigned a value and specifies a location in memory. Variables can be classified into various categories and subsets of the categories as shown in the diagram below. For example, any reference to a single numeric variable includes simple numerics and elements of numeric arrays.

- variable
  - single variable
    - numeric variable
      - simple numeric
      - array element
    - string variable
      - simple string
        - substring
      - string array element
        - substring
  - array variable
    - numeric array
      - matrix
      - vector
    - string array
      - matrix
      - vector

Word – Two bytes; a group of 16 binary digits (bits).

ASCII Character Codes

ASCII Char.	EQUIVALENT FORMS Binary	Oct	Hex	Dec
NULL	00000000	000	00	0
SOH	00000001	001	01	1
STX	00000010	002	02	2
ETX	00000011	003	03	3
EOT	00000100	004	04	4
ENQ	00000101	005	05	5
ACK	00000110	006	06	6
BELL	00000111	007	07	7
BS	00001000	010	08	8
HT	00001001	011	09	9
LF	00001010	012	0A	10
VT	00001011	013	0B	11
FF	00001100	014	0C	12
CR	00001101	015	0D	13
SO	00001110	016	0E	14
SI	00001111	017	0F	15
DLE	00010000	020	10	16
DC1	00010001	021	11	17
DC2	00010010	022	12	18
DC3	00010011	023	13	19
DC4	00010100	024	14	20
NAK	00010101	025	15	21
SYNC	00010110	026	16	22
ETB	00010111	027	17	23
CAN	00011000	030	18	24
EM	00011001	031	19	25
SUB	00011010	032	1A	26
ESC	00011011	033	1B	27
FS	00011100	034	1C	28
GS	00011101	035	1D	29
RS	00011110	036	1E	30
US	00011111	037	1F	31

ASCII Char.	EQUIVALENT FORMS Binary	Oct	Hex	Dec
space	00100000	040	20	32
!	00100001	041	21	33
"	00100010	042	22	34
#	00100011	043	23	35
$	00100100	044	24	36
%	00100101	045	25	37
&	00100110	046	26	38
'	00100111	047	27	39
(	00101000	050	28	40
)	00101001	051	29	41
*	00101010	052	2A	42
+	00101011	053	2B	43
,	00101100	054	2C	44
–	00101101	055	2D	45
.	00101110	056	2E	46
/	00101111	057	2F	47
0	00110000	060	30	48
1	00110001	061	31	49
2	00110010	062	32	50
3	00110011	063	33	51
4	00110100	064	34	52
5	00110101	065	35	53
6	00110110	066	36	54
7	00110111	067	37	55
8	00111000	070	38	56
9	00111001	071	39	57
:	00111010	072	3A	58
;	00111011	073	3B	59
<	00111100	074	3C	60
=	00111101	075	3D	61
>	00111110	076	3E	62
?	00111111	077	3F	63

ASCII Char.	EQUIVALENT FORMS Binary	Oct	Hex	Dec
@	01000000	100	40	64
A	01000001	101	41	65
B	01000010	102	42	66
C	01000011	103	43	67
D	01000100	104	44	68
E	01000101	105	45	69
F	01000110	106	46	70
G	01000111	107	47	71
H	01001000	110	48	72
I	01001001	111	49	73
J	01001010	112	4A	74
K	01001011	113	4B	75
L	01001100	114	4C	76
M	01001101	115	4D	77
N	01001110	116	4E	78
O	01001111	117	4F	79
P	01010000	120	50	80
Q	01010001	121	51	81
R	01010010	122	52	82
S	01010011	123	53	83
T	01010100	124	54	84
U	01010101	125	55	85
V	01010110	126	56	86
W	01010111	127	57	87
X	01011000	130	58	88
Y	01011001	131	59	89
Z	01011010	132	5A	90
[	01011011	133	5B	91
\	01011100	134	5C	92
]	01011101	135	5D	93
^	01011110	136	5E	94
_	01011111	137	5F	95

ASCII Char.	EQUIVALENT FORMS Binary	Oct	Hex	Dec
`	01100000	140	60	96
a	01100001	141	61	97
b	01100010	142	62	98
c	01100011	143	63	99
d	01100100	144	64	100
e	01100101	145	65	101
f	01100110	146	66	102
g	01100111	147	67	103
h	01101000	150	68	104
i	01101001	151	69	105
j	01101010	152	6A	106
k	01101011	153	6B	107
l	01101100	154	6C	108
m	01101101	155	6D	109
n	01101110	156	6E	110
o	01101111	157	6F	111
p	01110000	160	70	112
q	01110001	161	71	113
r	01110010	162	72	114
s	01110011	163	73	115
t	01110100	164	74	116
u	01110101	165	75	117
v	01110110	166	76	118
w	01110111	167	77	119
x	01111000	170	78	120
y	01111001	171	79	121
z	01111010	172	7A	122
{	01111011	173	7B	123
|	01111100	174	7C	124
}	01111101	175	7D	125
~	01111110	176	7E	126
DEL	01111111	177	7F	127

Use this table to determine what keys can be used with the CONTROL key to obtain a control code. First, find the desired code in the first column. Then read across that line to find the two or three keys which produce that character when pressed with CONTROL. For example, LF (linefeed) can be obtained by pressing CONTROL with one of the following:

- *
- J
- SHIFT J

The DEL character is the only one that can't be obtained using the CONTROL key.

Roman Extension Character Codes

ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
CLEAR	10000000	200	128
IV	10000001	201	129
BL	10000010	202	130
IV/BL	10000011	203	131
UL	10000100	204	132
IV/UL	10000101	205	133
BL/UL	10000110	206	134
IV/BL/UL	10000111	207	135
White	10001000	210	136
Red	10001001	211	137
Yellow	10001010	212	138
Green	10001011	213	139
Cyan	10001100	214	140
Blue	10001101	215	141
Magenta	10001110	216	142
Black	10001111	217	143
CLEAR	10010000	220	144
IV	10010001	221	145
BL	10010010	222	146
IV/BL	10010011	223	147
UL	10010100	224	148
IV/UL	10010101	225	149
BL/UL	10010110	226	150
IV/BL/UL	10010111	227	151
White	10011000	230	152
Red	10011001	231	153
Yellow	10011010	232	154
Green	10011011	233	155
Cyan	10011100	234	156

IV - Inverse video
BL - Blinking
UL - Underline

ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
Blue	10011101	235	157
Magenta	10011110	236	158
Black	10011111	237	159
	10100000	240	160
Á	10100001	241	161
Î	10100010	242	162
Ó	10100011	243	163
Ú	10100100	244	164
Â	10100101	245	165
Ê	10100110	246	166
Ô	10100111	247	167
´	00101000	250	168
`	00101001	251	169
^	00101010	252	170
¨	10101011	253	171
~	10101100	254	172
Ê	10101101	255	173
Ô	10101110	256	174
£	10101111	257	175
–	10110000	260	176
Ã	10110001	261	177
ã	10110010	262	178
°	10110011	263	179
Ç	10110100	264	180
ç	10110101	265	181
Ñ	10110110	266	182
ñ	10110111	267	183
¡	10111000	270	184
¿	10111001	271	185
¤	10111010	272	186
£	10111011	273	187
¢	10111100	274	188
§	10111101	275	189

ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
¢	10111110	276	190
•	10111111	277	191
â	11000000	300	192
ê	11000001	301	193
ô	11000010	302	194
û	11000011	303	195
á	11000100	304	196
é	11000101	305	197
ó	11000110	306	198
ú	11000111	307	199
à	11001000	310	200
è	11001001	311	201
ò	11001010	312	202
ù	11001011	313	203
ä	11001100	314	204
ë	11001101	315	205
ö	11001110	316	206
ü	11001111	317	207
Å	11010000	320	208
î	11010001	321	209
ø	11010010	322	210
Æ	11010011	323	211
å	11010100	324	212
í	11010101	325	213
ø	11010110	326	214
æ	11010111	327	215
Ä	11011000	330	216
ì	11011001	331	217
Ö	11011010	332	218
Ü	11011011	333	219
É	11011100	334	220
ï	11011101	335	221
ß	11011110	336	222

ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Decimal
	11011111	337	223
↑	11100000	340	224
┣	11100001	341	225
┏	11100010	342	226
┳	11100011	343	227
━	11100100	344	228
┓	11100101	345	229
┫	11100110	346	230
┃	11100111	347	231
╋	11101000	350	232
├	11101001	351	233
┌	11101010	352	234
┬	11101011	353	235
─	11101100	354	236
┐	11101101	355	237
┤	11101110	356	238
┼	11101111	357	239
→	11110000	360	240
┣	11110001	361	241
└	11110010	362	242
┴	11110011	363	243
┼	11110100	364	244
┘	11110101	365	245
┫	11110110	366	246
↓	11110111	367	247
←	11111000	370	248
┻	11111001	371	249
┗	11111010	372	250
┻	11111011	373	251
┣	11111100	374	252
┛	11111101	375	253
┳	11111110	376	254
▒	11111111	377	255

Decimal values 160 through 255 access the Nationalized and Drawing Characters.

Metric Conversion Table

English Units	Metric Units	To convert from English to Metric, multiply by:	To convert from Metric to English, multiply by:
Length			
mil	micrometre (micron)	2.54×10^{1}☆	$3.937\ 007\ 874 \times 10^{-2}$
inch	millimetre	2.54×10^{1}☆	$3.937\ 007\ 874 \times 10^{-2}$
foot	metre †	3.048×10^{-1}☆	3.280 839 895
mile (intl.)	kilometre	1.609 344☆	$6.213\ 711\ 922 \times 10^{-1}$
Area			
$inch^2$	$millimetre^2$	$6.451\ 6 \times 10^{2}$☆	$1.550\ 003\ 100 \times 10^{-3}$
$foot^2$	$metre^2$	$9.290\ 304 \times 10^{-2}$☆	$1.076\ 391\ 042 \times 10^{1}$
$mile^2$	$kilometre^2$	2.589 988 110	$3.861\ 021\ 585 \times 10^{-1}$
acre (U.S. survey)	hectare	$4.046\ 873 \times 10^{-1}$	2.471 044
Volume			
$inches^3$	$millimetres^3$	$1.638\ 706\ 4 \times 10^{4}$☆	$6.102\ 374\ 409 \times 10^{-5}$
$feet^3$	$metres^3$	$2.831\ 684\ 659 \times 10^{-2}$	$3.531\ 466\ 672 \times 10^{1}$
ounces (U.S. fluid)	$centimetres^3$	$2.957\ 353 \times 10^{1}$	$3.381\ 402 \times 10^{-2}$
gallon (U.S. fluid)	litre ‡	3.785 412	$2.641\ 721 \times 10^{-1}$
Mass			
pound (avdp.)	kilogram	$4.535\ 923\ 7 \times 10^{-1}$☆	2.204 622 622
ton (short)	ton (metric)	$9.071\ 847\ 4 \times 10^{-1}$☆	1.102 311 311
Force			
ounce (force)	dyne	$2.780\ 138\ 510 \times 10^{4}$	$3.596\ 943\ 090 \times 10^{-5}$
pound (force)	newton	4.448 221 615	$2.248\ 089\ 431 \times 10^{-1}$
Pressure			
psi	pascal	$6.894\ 757\ 293 \times 10^{3}$	$1.450\ 377\ 377 \times 10^{-4}$
inches of Hg (at 32°F)	millibar	$3.386\ 4 \times 10^{1}$	$2.952\ 9 \times 10^{-2}$
Energy			
BTU (IST)	Calorie (kg, thermochem.)	$2.521\ 644\ 007 \times 10^{-1}$	3.965 666 831
BTU (IST)	watt-hour	$2.930\ 710\ 702 \times 10^{-1}$	3.412 141 633
BTU (IST)	joule §	$1.055\ 055\ 853 \times 10^{3}$	$9.478\ 171\ 203 \times 10^{-4}$
ft•lb	joule	1.355 817 948	$7.375\ 621\ 493 \times 10^{-1}$
Power			
BTU (IST)/hr	watt	$2.930\ 710\ 702 \times 10^{-1}$	3.412 141 633
horsepower (mechanical)	watt	$7.456\ 998\ 716 \times 10^{2}$	$1.341\ 022\ 090 \times 10^{-3}$
horsepower (electric)	watt	7.46×10^{2}☆	$1.340\ 482\ 574 \times 10^{-3}$
ft•lb/s	watt	1.355 817 948	$7.375\ 621\ 493 \times 10^{-1}$
Temperature			
°Rankine	kelvin	1.8☆	$5.555\ 555\ 556 \times 10^{-1}$
°Fahrenheit	°Celsius	°C=(°F−32) /1.8☆	°F = (°C x 1.8)+32☆

☆ Exact conversion
† Conversion redefined in 1959
‡ Conversion redefined in 1964
§ Conversion redefined in 1956

Note: The preferred metric unit for force is the newton; for pressure, the pascal; and for energy, the joule.

Prefix	Symbol	Multiplier
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	k	10^{3}
hecto	h	10^{2}
deka	da	10^{1}

Prefix	Symbol	Multiplier
deci	d	10^{-1}
centi	c	10^{-2}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}

Sources

American Society for Testing and Materials (ASTM), "Standard for Metric Practice". Reprinted from Annual Book of ASTM Standards.

U.S. Department of Commerce, National Bureau of Standards, "NBS Guidelines for the Use of the Metric System". Reprinted from Dimensions/NBS. (October 1977).

Reset Conditions

The following table shows the status of various conditions when the indicated operations are performed.

	SCRATCHA or Power On (Value)	Reset	SCRATCH	RUN	CONT
Variables	R (none)	–	R	R	–
RESult	R (0)	–	–	–	–
Subroutine return pointers	R (none)	R	R	R	–
Angular units	R (RAD)	R	R	R	–
Numeric output mode	R (STANDARD)	R	R	–	–
Random number seed	R (π/180)	R	R	R	–
Standard printer	R (select code 16)	–	–	–	–
Printall printer	R (select code 16)	–	–	–	–
Standard mass storage device	R (:T15)	–	–	–	–
SFK definitions	R (Initial)	–	–	–	–
Processing mode	R (SERIAL)	–	R	–	–
Live keyboard mode	R (INTERACTIVE)	R	–	–	–
Binary routines	R (none)	–	–	–	–
Files table	R (none)	R	R	R	–
DATA pointers	R (none)	R	R	R	–
ERRL, ERRN	R (0,0)	R	R	R	–

– means unchanged

R means restored to power on values

Memory

This section delves into the structure, organization and use of User Read/Write Memory. It is not intended to be a complete explanation of memory, but to explain it as it relates to programming operations.

Read/Write Memory

The System 9845 Desktop Computer uses two types of memory: Read/Write Memory and Read Only Memory (ROM). Read/Write memory is used to store programs and data. When you store a program or data, you "write" into the memory. When you access a line of your program or a data element, you "read" from memory, thus the term Read/Write. Read/Write memory is temporary; it can be changed or erased. The contents of Read/Write memory are lost when the computer is shut off.

Programs and data in Read/Write Memory can be saved for future use by recording the information on a tape cartridge or other storage medium.

Memory Test and Loss

Each time the computer is switched on, the Read/Write memory is automatically tested. If a block of memory is found to be defective, you are warned with a message. This results in less memory that is available for your use. You can still use the computer and may not need the defective memory, depending on your application. To determine how much memory is available for your use, execute SCRATCH A, then LIST. This displays the amount in bytes. Any decrease in size would be in an increment of approximately 8 192 bytes. Call your HP Sales and Service Office for assistance.

Read Only Memory

Read Only Memory differs from Read/Write Memory in that it is permanent. When the computer is turned off, the Read Only Memory is unaffected. Each option ROM is inserted into one of the drawers in the sides of the machine, making it possible to expand the language and capabilities. A small amount of Read/Write Memory is used by some option ROMs. This area is called "working storage". The working storage used by each ROM is listed in the Installation, Operation, and Test Manual.

Conserving Memory

Large programs that involve large amounts of data can sometimes require more memory than is available for use. This section presents some ways to conserve memory usage when writing a program and using data.

One way to use less memory in a program is to limit the use of REM statements and comments in the program. This limits program readability and documentation, but does conserve memory usage and decrease program execution time.

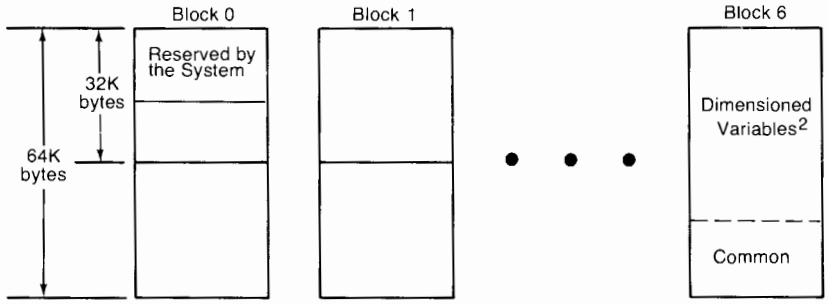
The use of subprograms can also conserve memory. Variables used within subprograms either share memory space with calling program variables or use memory only temporarily. So rather than creating new variables for various routines, thus using more memory, a subprogram can be used. In addition, the use of many short program segments, rather than a few large segments, results in better memory packing efficiency.

The use of SHORT and INTEGER precision variables, rather than full precision, is a good way to conserve memory in a program that has a great deal of data. This technique is most useful when dealing with large arrays. However, it has two limitations: all calculations are performed with full-precision accuracy, so INTEGER and SHORT precision variables must be converted before and after the operation. This slows down execution. Another limitation can arise when inverting a matrix that is not full precision; the results will almost never be entirely accurate due to rounding errors during calculation.

A fourth way to conserve memory is to break a program down into several sections and SAVE each section in a different file. This is known as **overlaying**. Each section can be brought into memory using LINK. This operation preserves the values of variables, but erases each section of the program as another one is linked in.

Memory Organization

Read/Write Memory is divided into blocks. Blocks are 64K bytes long. The following diagram illustrates the blocks of memory. There can be up to seven blocks[1], depending on the amount of memory installed in your computer.



1 Odd-numbered memory blocks are used by the operating system and are not part of User Read/Write Memory.

2 The amount of memory taken by dimensioned variables can change depending on your program.

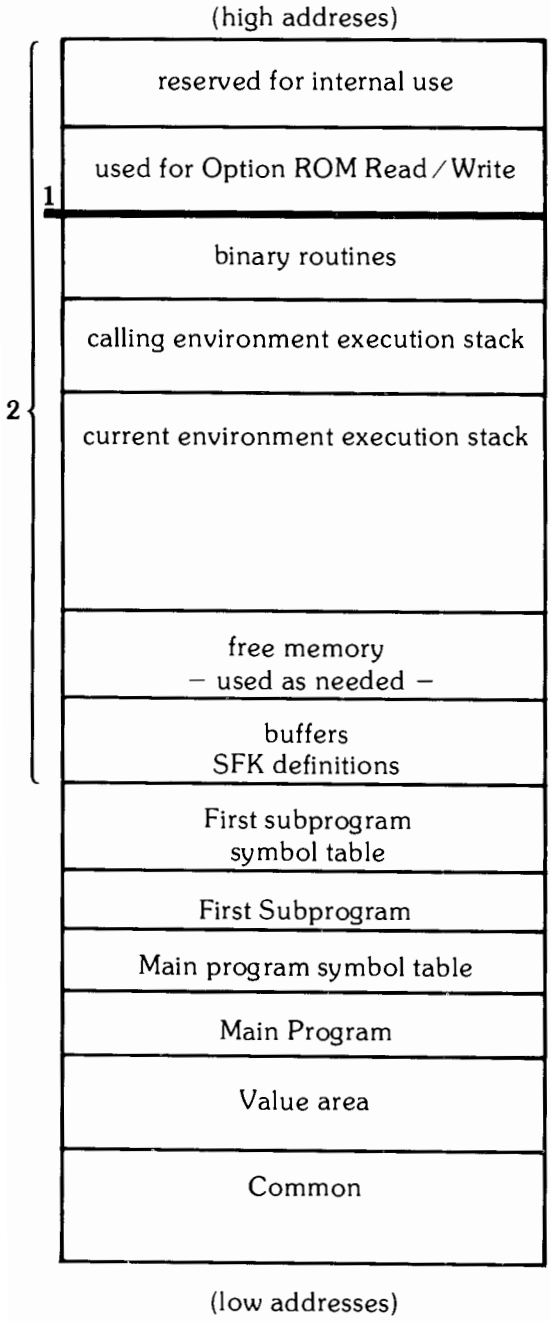
The division of memory into blocks imposes limitations on programs and variables. The limitations are –

- No main program or subprogram can be larger than one block of memory. A 1000-line program typically fills one half of a 64K block.
- No main program or subprogram can cross a block boundary. That is, the main or subprogram must be contained entirely in one block of memory.

  This limitation may cause you to get an unexpected memory overflow error, ERROR 2, though executing LIST indicates there is ample memory available. The reason for this is that the available memory is not all in the same block.

  To avoid this situation, it is advisable to organize your program into a short main program and a series of short subprograms, rather than use long program segments. This works well because a block can contain more than one subprogram. Additionally, a program can consist of a main program and subprograms in several different blocks.
- No simple numeric or string variable or array element can cross a block boundary. Arrays of long strings and long simple strings can cause an unexpected memory overflow or waste large amounts of memory. For example, suppose you are allocating memory to some variables in a DIM or COM statement. Suppose that there is a 25K-byte string following a numeric variable, but only 10K bytes left in the block after the numeric variable is allocated memory space. The string will have to be stored in the next block, thus wasting 10K bytes of memory. Thus, the order of large strings in DIM or COM statements can affect the amount of memory needed to run a program.
- Each time an array crosses a block boundary, six bytes of memory are added to the total amount needed to store the array.
- The execution stack and any binary routines must be contained in block 0. You could get a memory overflow when the other blocks are not full if the execution stack gets too large. This can be caused by recursive subprogram calls and intermediate results involving long strings. Some program restructuring may be necessary.

Simplified Read/Write Memory Organization



This area is used for system configuration information – printer select code, for example.

The amount used by each ROM is listed in the Installation, Operation, and Test Manual.

Binary routines are added to existing ones as they are loaded into memory using LOAD BIN.

The execution stacks are the way the computer keeps track of where program execution is. They contain DATA pointers, subroutines return pointers, FOR-NEXT matching, and other indicators for program execution. The current environment execution stack also contains a program pointer to monitor which line is being executed currently. The size of an execution stack varies during program execution.

Buffers for I/O and mass storage operations use Read/Write Memory. SFK defintions use 160 (138 if there is no lefthand tape drive) bytes at power-on.

Each symbol table contains variable names, any variable attributes (integer precision, array, etc.), and a value pointer which points to the value of the variable.

Each successive subprogram and its symbol table comes "after" (have a higher address than) the previous one.

Contains the values for all main and subprogram variables.

Contains the values of all variables declared in COM statements.

1 This boundary is fixed at power-on.
2 This information must be in Block 0.

DMA and FHS NOFORMAT Transfers

The purpose of this section is to explain the organization of User Read/Write memory as it relates to DMA and FHS NOFORMAT transfer rates.

If a DMA or FHS NOFORMAT transfer is to achieve the maximum transfer rate, the variable that the data is entered into or output from must reside totally within one memory block. The following guidelines ensure that a variable used for a DMA or FHS NOFORMAT transfer resides totally within one memory block.

- The variable is a simple string or simple numeric variable, or
- The variable is a real, short, or integer numeric array that uses less than 32 720 words[1] of memory. The variable MUST be the first variable declared in the main program. There must be NO memory declared for use by a special ROM (e.g. Assembly Development ROM).

The following is useful in determining the size and impact of variable allocation.

- No simple variable or array element can cross a block boundary. Arrays of long strings and long simple strings can also cause an unexpected memory overflow or waste large amounts of memory. For example, suppose you are allocating memory to some variables in a DIM or COM statement. Suppose that there is a 25K byte character string following a numeric variable, but only 10K bytes left in the block after the numeric variable is allocated memory space. The string will have to be stored in the next block, thus wasting 10K bytes of memory. Thus, the order of large strings in DIM or COM statements can affect the amount of memory needed to run a program.
- Each time an array crosses a block boundary, six bytes of memory are added to the total amount needed to store the array.

Storage of Variables

To determine how many bytes variables require when stored in memory, use the following tables.

Simple Variable	Amount of Memory Used
Full precision	10 bytes
Short precision	6 bytes
Integer	4 bytes
String	6 bytes + length (1 byte per character, rounded up to an even integer)

Array Variable	Amount of Memory Used
Full precision	10 bytes + 4 bytes per dimension + 8 bytes per element
Short precision	10 bytes + 4 bytes per dimension + 4 bytes per element
Integer	10 bytes + 4 bytes per dimension + 2 bytes per element
String	12 bytes + 4 bytes per dimension + 2 bytes per element + length of each string (1 byte per character, rounded up to an even integer)

1 Note that array overhead takes up to 17 words for one-block arrays.

System 45 Compatibility

HP Compatible BASIC

The BASIC language as implemented on your HP 9845 is an enhanced form of HP Compatible BASIC. HP Compatible BASIC consists of statements, functions, operators, and commands that are implemented in HP BASIC machines. HP Compatible BASIC is implemented on the HP 9845 as Level I. Level I refers to the highest performance computational products. Thus, any program consisting entirely of Level I BASIC language can be transported to any Level I BASIC machine.

Below is a list of HP Level I BASIC. Contact your HP Sales and Service Office to obtain information concerning the transporting of programs between machines.

Operators

+	=	AND
−	>	OR
*	<	NOT
/	≥	EXOR
^	≤	
DIV	< >	&
MOD		

Functions

ABS	ERRL	LWC$	LIN
EXP	ERRN	REV$	SPA
INT	ERRM$	UPC$	TAB
LGT		CHR$	PAGE
LOG	COL	LEN	
MAX	DET	NUM	SIN
MIN	DOT	POS	COS
RND	ROW	RPT$	TAN
SGN	SUM	TRIM$	ASN
DROUND		VAL	ACS
PROUND		VAL$	ATN
FRACT	TYP		
PI			
SQR			

Statements

ASSIGN	
BEEP	
CALL	
COM*	
COPY	
CREATE	GOTO
DATA	GRAD
DEF FN	IF
DEG	INPUT
DIM	INTEGER
EDIT	LET
END	LINPUT
FIXED	MAT array = array
FLOAT	MAT array = array + − * / . = < > # < ≤ > ≥ array
FN END	MAT...CON
FOR	MAT array = (num. exp.)
NEXT	MAT array = (num. exp.) + − * / = < > # < ≤ > ≥ array
GOSUB	MAT array = array + − * / = < > # < ≤ > ≥ (num. exp.)

* The type words INTEGER, SHORT and REAL are the only ones which can be specified in a COM statement or formal parameter list. Arrays are limited to 6 dimensions.

MAT...INV
MAT...TRN
MAT...IDN
MAT...ZER
MAT...CSUM
MAT... RSUM
MAT INPUT
MAT PRINT
MAT PRINT#
MAT READ
MAT READ#
OFF END
OFF ERROR
ON END
ON ERROR
ON...GOSUB
ON...GOTO
OPTION BASE
PAUSE
PRINT
PRINT#
PRINT USING
PURGE
RAD
RANDOMIZE
READ
READ#
REAL
REDIM
REM
RESTORE
RETURN
SHORT
STANDARD
STOP
SUB
SUBEXIT
SUBEND
WAIT

9845A vs. 9845B/C

Listed below are differences between the 9845A and 9845B/C which affect operating and programming:

- The operating system of the 9845B/C is internal, leaving all eight slots in each ROM drawer free for option ROMs.
- The range of memory sizes on the 9845A is 16–64 K-bytes. This is increased on the 9845B/C to approximately 56–449 K-bytes.
- The greatest line number available on the 9845A is 9999. The greatest line number available on the 9845B/C is 32 766.
- The size of the recall buffer is increased from 344 bytes (on the 9845A) to 1296 bytes (on the 9845B/C).
- When using a tape cartridge for mass storage operations, the directory of the tape is stored in Read/Write Memory of the 9845B/C. This reduces tape wear because the directory on the tape doesn't have to be accessed every time a tape read or write operation takes place.
- Four additional syntax were added to the 9845B/C: CAT TO, ON KBD, OFF KBD, and KBD$.
- On the 9845B/C, all alternate characters, both nationalized and drawing, are accessed using the CHR$ function. On the 9845A, only some of the nationalized characters are available, and are accessed using the shift-in and shift-out control codes.
- The only types of mass storage files which are compatible between the 9845A and the 9845B/C are the DATA and BDAT files. The 9845B/C cannot interpret any other 9845A files.
- Programs that contain a large number of binary routines or recursive algorithms may cause a memory overflow error when run on the 9845B/C, although they don't on the 9845A.
- LOAD ALL is programmable on the 9845B/C. It is not on the 9845A.
- The printout area of the CRT on the 9845B/C can be addressed directly. A complete description of this capability appears in Appendix A of the BASIC Programming manual.
- Some of HP's software will run only on particular models of the System 45. If you have questions about software compatibility, refer to the System 45 Pricing Information brochure (P/N 5953-4572D) or call your HP Sales and Service Office.

File Compatibility

The 9845B/C can interpret mass storage files created on other HP desktop computers. The files it can interpret are:

- DATA and BDAT type files from the System 35 or the 9845A.
- KEYS files from the System 35.

9845A Graphics ROM vs. 9845B Graphics ROM

9845B graphics differs from 9845A graphics in the following ways:

- CSIZE incorporates a character slant as its third parameter.
- GLOAD and GSTORE reside in the 9845B Graphics ROM and in the 9845A Mass Storage ROM.
- LDIR and PDIR both incorporate a run, rise parameter as a means of specifying an angle.
- The line types are slightly different. The 9845B has a line type 10 with a major tick mark.
- When using the LABEL or LETTER statement, the linefeed distance on the 9845B/C is 15/16 of the distance of a linefeed on the 9845A. This is one dot for default character size.

9845B Graphics ROM vs. 9845C Graphics ROM

Listed below are differences between the 9845B Graphics ROM (for the 9845B Model 150 or 190) and the 9845C Graphics ROM:

- With the 9845C Graphics ROM, the CURSOR and DIGITIZE statements use the ARROW KEYS as the default graphics input device, rather than the plotter as with the 9845B Graphics ROM. The status strings for these statements also differ between ROMs. Refer to the CURSOR statement in the Color Graphics manual for more specific information.
- When using the 9845C Graphics ROM, and both the ALPHA and GRAPHICS statements are enabled, it may be difficult to tell which characters are PRINTed (alpha raster) and which characters are LABELed or LETTERed (graphics raster) when the CSIZE is at default (3.3 GDUs).
- GCLEAR with the 9845C Graphics ROM clears only the soft clip area of graphics memory for all active memory planes. GCLEAR with the 9845B Graphics ROM clears the entire graphics memory.
- When using GLOAD and GSTORE, the System 45C needs 3*(455 rows)*(35 words per row) elements to store the entire graphics memory contents. The contents are stored as a word of memory 1, a word of memory 2, and a word of memory 3. The System 45B Model 150/190 needs (455 rows)*(35 words per row) + 1(pointer) to store the complete graphics memory contents. Refer to the GLOAD or GSTORE statements in the Color Graphics manual for exact array sizes.
- The 9845C Graphics ROM uses eight pen numbers, while the 9845B Graphics ROM uses five. The difference is most noticeable when using an external, multi-pen plotter. A PEN 5 statement is interpreted as PEN 0 with the 9845B Graphics ROM and as PEN 1 with the 9845C Graphics ROM. A negative pen always acts as PEN −1 with the 9845B Graphics ROM. A negative pen with the 9845C Graphics ROM erases different memory planes, depending upon the pen number. Refer to the PEN statement in the Color Graphics manual for more information.

- With the 9845B Graphics ROM, specifying a second plotter turns off the first plotter. With the 9845C Graphics ROM, specifying a second plotter leaves both plotters active. The plotters remain active until they are turned off by a PLOTTER IS OFF statement.
- When using the 9845B Graphics ROM, the cursor type is determined by: Cursor Type MOD 2. If the result is 0, the small blinking cross is used. If the result is 2, the large full screen cross-hair is used. When using the 9845C Graphics ROM, the cursor type is determined by: Cursor Type MOD 4. If the result is 0, the pointer is not affected by any other graphics statements until another POINTER statement with a different cursor type is executed. If the result is 1, a full screen cross-hair is displayed. If the result is 2, the small cross is displayed. If the result is 3, the small blinking alpha cursor is displayed.
- When using the 9845B Graphics ROM, the status string for WHERE is one character (pen up/down). The status string for the 9845C Graphics ROM is three characters (pen up/down and region). Refer to the WHERE statement in the Color Graphics manual for more information.
- A text string highlighted using the 9845B Graphics ROM results in only the first character being highlighted when transported to a System 45C with the 9845C Graphics ROM.
- The Katakana character set on the System 45C contains a yen sign instead of the backslash character on the System 45B.

9845C Graphics ROM vs. Enhanced Graphics ROM

Listed below are differences between the 9845C Graphics ROM (for the System 45C) and the Enhanced Graphics ROM (for the System 45B Model 200/250/290 or the System 45C):

- The Enhanced Graphics ROM provides the following capabilities which are not provided by the 9845C Graphics ROM:
  - Rubber banding (POINTER type = 4, 5, or 6)
  - Fast tracking (GRAPHICS INPUT IS TABLET)
  - Fast alpha
- The Enhanced Graphics ROM provides some capabilities that work only on a System 45B Model 200/250/290 and not on a System 45C (refer to the Monochromatic Graphics manual for more information):
  - Fast erasing
  - Arcs and circles
  - Rubber banding with no background loss
- Programs which have been STOREd on a computer which has the 9845C Graphics ROM can be LOADed on a computer which has the Enhanced Graphics ROM, and vice versa. The only exceptions in program execution are:
  - "type" = 4, 5, or 6 in POINTER statement
  - "graphics input identifier string" = TABLET in the GRAPHICS INPUT IS statement
  - PLOTTER IS HPGL specifies 8 pens with Enhanced Graphics ROM and 4 pens for 9845C Graphics ROM
  - GLOAD and GSTORE do not work between color and monochromatic CRTs.
- You can LOAD ALL any STORE ALL files between the System 45B Model 200/250/290 and the System 45C, as long as both computers have the Enhanced Graphics ROM, the same options, and the same memory sizes.
- The two ROMs give different results if you use alphanumeric highlights with screen addressing and overprinting in your programs.

- The following keywords have no effect when executed on a System 45B Model 200/250/290:
  - DEGAUSS
  - CONVERGE
  - MEMORY
  - GSTAT(5,6, or 7)
- The Monochromatic Graphics manual explains how statements which specify colors and memory planes are executed on a System 45B Model 200/250/290 (monochromatic CRT).

Computer Museum

Standard Processor vs. Enhanced Processor

The HP 9845B/C Models 200, 250, and 290 are equipped with an "enhanced processor". The HP 9845B/C models 100, 150, and 190 are equipped with the "standard processor".

The increased speed of the enhanced processor is due to "microcoding" the parts of the operating system where the language processor spends most of its time for a typical computational program. These microcode routines run from five to fifteen times faster than the standard processor routines. The result is an overall speed increase of about three times, depending on the particular program. (The enhanced processor has very little effect on assembly language programs or I/O-intensive programs.)

Most of the microcode routines affect the Arithmetic Logic Unit (ALU) of the language processor. Thus, speed increases are most noticeable when a program is spending almost all of its time computing and there are no I/O or assembly language operations being done.

Because of differences in program structure and application, it is difficult to determine the speed increase which you may see with the enhanced processor. However, it is possible to get an idea of how much faster the individual routines run. The System 45 Computer Specifications brochure includes run times for the math and trigonometric routines (for both the standard and enhanced processors).

Maximizing Performance

There are some general guidelines to follow to make your programs run as fast as possible, regardless of the processor that your desktop computer has. Here are a few of them:

- Use full-precision (REAL) variables for math operations.
- Use integer-precision (INTEGER) variables in FOR-NEXT loops.
- Use integer-precision (INTEGER) variables for array subscripts.
- Do not use short-precision (SHORT) variables.
- Use GOSUBs instead of CALLs whenever possible.
- Turn off TRACE and TRACK when not needed.

Graphics Firmware Differences

Statement	Graphics ROM documented in manual HP p/n 09845-91050 HP p/n 09845-91051	9845C Graphics ROM documented in manual HP p/n 09845-92050	Enhanced Graphics ROM documented in manual HP p/n 09845-93050 HP p/n 09845-92051
ALPHA	Not implemented	Enables alphanumeric area	Enables alphanumeric area
AREA COLOR	Not implemented	Selects fill from Color Cylinder model	On 9845B, uses only the luminosity value for fill. On 9845C, works the same as 9845C Graphics ROM.
AREA INTENSITY	Not implemented	Selects fill from R,G,B cube color model.	On 9845B, uses the largest of the three values for fill. On 9845C, works the same as 9845C Graphics ROM.
AXES	No difference	No difference	No difference
CLIP	No difference	No difference	No difference
CONVERGE	Not implemented	Allows convergence to be performed	On 9845B, not implemented. On 9845C, works the same as 9845C Graphics ROM.
CSIZE	No difference	No difference	No difference
CURSOR	Status string is 1 character long	Status string is maximum of 40 characters. HP 9111 string is same as HP 9874. Additional clipping and tracking information added to all status strings.	Status string is maximum of 40 characters. HP 9111 has different string contents. Additional clipping and tracking information added to all status strings.
DEGAUSS	Not implemented	Performs CRT degaussing	On 9845B, not implemented. On 9845C, works the same as 9845C Graphics ROM.
DIGITIZE	Status string is 1 character long	Status string is maximum of 40 characters. HP 9111 string is same as 9874 characters 1-6 Additional clipping and tracking information added to all status strings.	Status string is maximum of 40 characters. HP 9111 string is same as 9874 characters 1-6. Characters 7-8 are menu item key values. Additional clipping and tracking information added to all status strings.
DRAW	No difference	No difference	No difference

Statement	Graphics ROM documented in manual HP p/n 09845-91050 HP p/n 09845-91051	9845C Graphics ROM documented in manual HP p/n 09845-92050	Enhanced Graphics ROM documented in manual HP p/n 09845-93050 HP p/n 09845-92051
DUMP GRAPHICS	External devices not allowed	External devices allowed with # specifier	External devices allowed with # specifier.
EXIT ALPHA	Not implemented	Disables alphanumeric area	Disables alphanumeric area
EXIT GRAPHICS	No difference	No difference	No difference
FRAME	No difference	No difference	No difference
GCLEAR	Clears entire CRT graphics memory	Clears only the CRT graphics memory within the soft clip area	Clears only the CRT graphics memory within the soft clip area
GLOAD	16 381 elements required for full-screen display Column and row pointer is NOT allowed	47 775 elements required for full-screen display Column and row pointer is allowed	On 9845B, 15 925 elements required for full-screen display Column and row pointer is allowed On 9845C, works the same as 9845C Graphics ROM.
GRAPHICS	No difference	No difference	No difference
GRAPHICS INPUT IS	Not implemented	Allows as devices array name (*) ARROW KEYS DIGITIZER HPGL LIGHT PEN	Allows as devices array name (*) ARROW KEYS DIGITIZER HPGL LIGHT PEN TABLET
GRAPHICS INPUT ...IS OFF	Not implemented	Deactivates device	Deactivates device
GRAPHICS INPUT ...IS ON	Not implemented	Activates device	Activates device
GRID	No difference	No difference	No difference
GSTAT	Not implemented	Returns value based upon index	Returns value based upon index
GSTORE	16 381 elements required for full-screen display Column and row pointer is NOT allowed	47 775 elements required for full-screen display Column and row pointer is allowed	On 9845B, 15 925 elements required for full-screen display Columna and row pointer is allowed On 9845C, works the same as 9845C Graphics ROM.
IPLOT	No difference	No difference	No difference
KEY LABELS	Not implemented	Transfers softkey labels to a string variable	Transfers softkey labels to a string variable
LABEL	No difference	No difference	No difference
LABEL KEY #	Not implemented	Labels specified softkey with string expression	Labels specified softkey with string expression

Statement	Graphics ROM documented in manual HP p/n 09845-91050 HP p/n 09845-91051	9845C Graphics ROM documented in manual HP p/n 09845-92050	Enhanced Graphics ROM documented in manual HP p/n 09845-93050 HP p/n 09845-92051
LABEL KEYS	Not implemented	Labels all softkeys with string expression	Labels all softkeys with string expression
LABEL USING	No difference	No difference	No difference
LAXES	Not implemented	Draws axes with labeled major tick marks	Draws axes with labeled major major tick marks
LDIR	No difference	No difference	No difference
LETTER	No difference	No difference	No difference
LGRID	Not implemented	Draws grid with labeled major tick marks	Draws grid with labeled major tick marks
LIMIT	No difference	No difference	No difference
LINE TYPE	No difference	No difference	No difference
LOCATE	No difference	No difference	No difference
LORG	No difference	No difference	No difference
MAT AIPLOT	Not implemented	Plots array contents as X,Y increments to current pen position in APUs	Plots array contents as X,Y increments to current pen position in APUs
MAT APLOT	Not implemented	Plots array contents as X,Y coordinates in APUs	Plots array contents as X,Y coordinates in APUs
MAT ARPLOT	Not implemented	Plots array contents as X,Y coordinates relative to an origin in APUs	Plots array contents as X,Y coordinates relative to an origin in APUs
MAT IPLOT	Not implemented	Plots array contents as X,Y increments to current pen position in current units. Optional FILL	Plots array contents as X,Y increments to current pen position in current units. Optional FILL
MAT PLOT	Not implemented	Plots array contents as X,Y coordinates in current units. Optional FILL	Plots array contents as X,Y coordinates in current units. Optional FILL
MAT RPLOT	Not implemented	Plots array contents as X,Y coordinates relative to an origin in current units. Optional FILL	Plots array contents as X,Y coordinates relative to an origin in current units. Optional FILL
MAT SYMBOL	Not implemented	Plots array contents as a specially defined labeled character with optional FILL	Plots array contents as a specially defined labeled character with optional FILL
MEMORY	Not implemented	Assigns specified color to the memory plane	On 9845B, no effect on program execution. On 9845C, works the same as 9845C Graphics ROM
MOVE	No difference	No difference	No difference
MSCALE	No difference	No difference	No difference

Statement	Graphics ROM documented in manual HP p/n 09845-91050 HP p/n 09845-91051	9845C Graphics ROM documented in manual HP p/n 09845-92050	Enhanced Graphics ROM documented in manual HP p/n 09845-93050 HP p/n 09845-92051
OFF GKEY	Not implemented	Disables end-of-line branch when button is pressed on the graphics input device	Disables end-of-line branch when button is pressed on the graphics input device
ON GKEY	Not implemented	Enables end-of-line branch when button is pressed on the graphics input device	Enables end-of-line branch when button is pressed on the graphics input device
PDIR	No difference	No difference	No difference
PEN	Selects pen from stall based on a modulo operation (pen 5 = 0 pen 9 = 4)	Selects pen from stall based on a modulo operation (pen 5 = pen 1 pen 9 = pen 1)	Selects pen stall based on the PLOTTER IS identifier string 9872A implies 4 pen plotters (pen 5 = pen 1 pen 9 = pen 1) HPGL implies 8 pen plotters (pen 5 = pen 5 pen 9 = pen 1)
PENUP	No difference	No difference	No difference
PLOT	No difference	No difference	No difference
PLOTTER IS	Allows only GRAPHICS 9872A INCREMENTAL as identifier strings Deactivates any active plotter. Multiple plotters are not allowed	Allows only GRAPHICS 9872A INCREMENTAL HPGL array name (*) as identifier strings Also allows individual memory planes as plotters Does NOT deactivate any active plotters. Multiple plotters are allowed	Allows only GRAPHICS 9872A INCREMENTAL HPGL array name (*) as identifier strings Also allows individual memory planes as plotters, but on 9845B the plots are replotted onto the one memory. Does NOT deactivate any active plotters. Multiple plotters are allowed
PLOTTER...IS OFF	No difference	No difference	No difference
PLOTTER...IS ON	Turns off previous plotter	Does not turn off previous plotter	Does not turn off previous plotter

Statement	Graphics ROM documented in manual HP p/n 09845-91050 HP p/n 09845-91051	9845C Graphics ROM documented in manual HP p/n 09845-92050	Enhanced Graphics ROM documented in manual HP p/n 09845-93050 HP p/n 09845-92051
POINTER	Type is either large (odd value) or small blinking (even value)	Type can be any of four: 0 = off 1 = large 2 = small 3 = underline remaining values are taken modulo 4 color of marker can also be specified	Type can be any of seven: 0 = off 1 = large 2 = small 3 = underline 4 = Rubber Band for memory 1 5 = Rubber Band for memory 2 6 = Rubber Band for memory 3 remaining values are taken modulo 7 color of marker can also be specified for 9845C.
POLYGON	Not implemented	Draws regular polygon, with approximations for circles. Optional FILL	Draws regular polygon. On 9845B, figures between 60 and 32 767 sides are drawn with hardware arc generator. Optional FILL
RATIO	No difference	No difference	No difference
RECTANGLE	Not implemented	Draws rectangle with optional FILL	Draws rectangle with optional FILL
SCALE	No difference	No difference	No difference
SETGU	No difference	No difference	No difference
SETUU	No difference	No difference	No difference
SHOW	No difference	No difference	No difference
TRACK...IS OFF	Not implemented	Stops tracking on specified plotter	Stops tracking on specified plotter
TRACK...IS ON	Not implemented	Enables plotter's marker to echo the movement of the graphics input device's cursor. Tracking to an array is allowed	Enables plotter's marker to echo the movement of the graphics input device's cursor. Tracking to an array is allowed
UNCLIP	No difference	No difference	No difference
WHERE	Status string is 1 character long	Status string is 3 characters long	Status string is 3 characters long

The other differences between the computers are:

1. A text string highlighted on the System 45B and then transported to a System 45C results in only the first character being highlighted on the System 45C.
2. The Katakana character set on the Enhanced Graphics ROM and the 9845C Graphics ROM contain yen signs instead of the backslash character on the 9845B Graphics ROM.

Computer Museum

Error Messages

1	Missing ROM or configuration error. Also, check to see if all option ROMs are installed properly. Perform the System Exerciser if the problem persists.
2	Memory overflow; subprogram larger than block of memory. Also check to see if your arrays are too large to fit in memory. If you are programming in assembly language, you may have specified an ICOM which is too large for your current availabe space.
3	Line not found or not in current program segment. Check the spelling of line labels and line identifiers.
4	Improper return. Branched into the middle of a subroutine.
5	Abnormal program termination; no END or STOP statement.
6	Improper FOR/NEXT matching.
7	Undefined function or subroutine. Check spellings.
8	Improper parameter matching. Check the parameter lists in SUB and CALL, and DEF FN and FN statements to see if they match in number and type.
9	Improper number of parameters. Check the number of arguments used in an FN or CALL reference. In assembly language, the number of arguments pass by an ICALL statement exceeds the number of parameter declarations in the subroutine entry section.
10	String value required.
11	Numeric value required.
12	Attempt to redeclare variable. Once a variable name has been declared in a DIM, COM, REAL, SHORT or INTEGER statement, it can't be redeclared in that program segment.
13	Array dimensions not specified. You must dimension the array, either explicitly or implicitly.
14	Multiple OPTION BASE statements or OPTION BASE statement preceded by variable declarative statements.
15	Invalid bounds on array dimension or string length in DIM, COM, REAL, SHORT or INTEGER statement. Strings can't be longer than 32 767 characters. The range of array subscripts is −32 767 through 32 767.
16	Dimensions are improper or inconsistent; more than 32 767 elements in an array. Check for wrong number of subscripts in an array reference. Check any matrix multiplication for proper sizes. An FREAD operation requires a receiving array to have the same number of dimensions as the array stored in the BDAT file, and that the number of elements be sufficient to hold the entire data. String length (in the case of string arrays) must also be consistent. Check the current dimensions of the receiving array in the program.

17	Subscript out of range.
18	Substring out of range or string too long. Check substring specifiers against length of string.
19	Improper value. Check numbers being entered, especially their exponents.
20	Integer precision overflow. The range is −32 768 through 32 767. An expression used in the syntax in one of your statements was out of range when rounded to an integer. Check the values of the variables used.
21	Short precision overflow. Short-precision numbers have six significant digits and an exponent in the range −63 through 63.
22	Real precision overflow. Full-precision numbers have twelve significant digits and an exponent in the range −99 through 99.
23	Intermediate result overflow.
24	TAN (n*π/2), when n is odd.
25	Magnitude of argument of ASN or ACS is greater than 1.
26	Zero to negative power.
27	Negative base to non-integer power.
28	LOG or LGT of negative number.
29	LOG or LGT of zero.
30	SQR of negative number.
31	Division by zero; or X MOD Y with Y = 0.
32	String does not represent valid number or string response when numeric data required. Check any use of VAL function and its argument. Check for correct spelling of variable name.
33	Improper argument for NUM, CHR$, or RPT$ function.
34	Referenced line is not IMAGE statement. Check the line identifier in the PRINT USING statement.
35	Improper format string.
36	Out of DATA. Make sure READ and DATA statements correspond. Use RESTORE if appropriate.
37	EDIT string longer than 160 characters. Try using a substring.
38	I/O function not allowed. TYP and other I/O functions aren't allowed in any I/O statement like DISP or PRINT. Place the value into a variable.
39	Function subprogram not allowed. An FN reference isn't allowed in any I/O statement, or in redim subscripts. Place the value into a variable.
40	Improper replace, delete or REN command. SUB and DEF FN can only be replaced by another SUB or DEF FN. They can only be deleted if the rest of the corresponding subprogram is deleted. A renumbering may cause out-of-range line numbers if completed, so an error occurs; check increment value.
41	First line number greater than second.
42	Attempt to replace or delete a busy line or subprogram. Typically, this is caused by trying to delete an input statement that is still requesting values.

43 Matrix not square. The dimensions of an identity matrix or of one used to find an inverse or determinant must be the same size.

44 Illegal operand in matrix transpose or matrix multiply. The result matrix can't be named the same as one of the operands.

45 Nested keyboard entry statements.

46 No binary in memory for STORE BIN or no program in memory for SAVE. There are no binaries currently in memory or there are no program lines in memory between the limits you specified. Check line numbers in SAVE against program in memory.

47 Subprogram COM declaration is not consistent with main program. Check number, type and dimensions of variables.

48 Recursion in single-line DEF FN function. Only subprograms can be called recursively.

49 Line specified in ON declaration not found.

50 File number less than 1 or greater than 10.

51 File not currently assigned. Execute an ASSIGN statement for the file, or check the accuracy of the file number used. LOAD ALL destroys all prior assignments.

52 Improper mass storage unit specifier. Check the values of the select code, unit code and controller address.

53 Improper file name. A file name can have 1-6 characters and can't contain a colon, quote mark, NULL or CHR$(255).

54 Duplicate file name. Choose another name or PURGE the old one. In some instances, RE-SAVE or RE-STORE may be an alternative statement to use.

55 Directory overflow. There is a maximum number of files that a mass storage medium can hold. A file will have to be removed to add another.

56 File name is undefined. Check the spelling.

57 Mass Storage ROM is missing. Check to see that the ROM is installed properly. Perform the System Exerciser if the problem persists.

58 Improper file type. Use LOAD for PROG files, ASSIGN and GET for DATA files, and LOADKEY for KEYS files.

59 Physical or logical end-of-file found. Attempting to READ# or PRINT# past the end of the file. If you are in serial mode, you have run out of data. If you are in random mode, you are reading a record beyond the reserved file length, or specified record number is too large. Compare the data list to the file size. This error can be trapped with the ON END statement.

60 Physical or logical end-of-record found in random mode. Attempting to read more data out of a single defined record than are actually there. Compare the data list to the record size.

61 Defined record size is too small for data item. You can either PURGE and CREATE the file with longer records or regroup the data being recorded.

62 File is protected or wrong protect code specified. Check to see that the protect code is included and spelled properly.

63 The number of physical records is greater than 32 767. That's the limit; use something smaller.

64 Medium overflow (out of user storage space). A file can't be set up because there isn't enough contiguous storage capacity on the medium. Use another medium, PURGE some of the files on the present medium, or try to repack (as explained in the Mass Storage ROM Manual.)

65 Incorrect data type. Each file must be read from the medium in the same way it was written. Check to make sure that you are not trying to retrieve a numeric data file as a program (with GET or LINK). This error may also occur if the wrong type of array variable is used when reading a BDAT-type file. Use TYP to find out what kind of data the computer is trying to read.

66 Excessive rejected tracks during a mass storage initialization. The medium can't be initialized. Medium wear on flexible discs, or a marginally-performing hard disc drive is usually the reason for this. With hard discs, it is recommended that the appropriate disc diagnostic tests be run for the drive involved. (Consult the operating and installation manual for the drive.) If the medium is a flexible disc, use a different one.

67 Mass storage parameter less than or equal to 0. Check values of variables. Record numbers, record lengths and number of defined records must be positive numbers.

68 Invalid line number in GET or LINK operation. This occurs only if the first line of a DATA-type file does not begin with a line number. This error should be expected only if the DATA file does not contain a program, or if the program in the file was created by another program and not by a SAVE or RE-SAVE instruction.

69 Format switch on the disc off. Turn it on for disc initialization to proceed.

70 Not a disc interface, or wrong device type. You are either using the wrong msus or are trying to reference the wrong type of device. Check for correspondence of device types. If they check out, your interface may be malfunctioning.

71 Disc interface power off. Turn it on. If it is already on, the interface may be malfunctioning.

72 Incorrect controller address or controller power off. If the former, change the address setting or change the program reference.

Additionally, this error can occur if the disc drive (7908 disc drive only) is temporarily occupied servicing a disc drive front panel request such as LOAD (cartridge), UNLOAD, STORE or RESTORE.

If all is in order, suspect that the controller or its interface is malfunctioning. See the peripheral's manual for location of the address switches.

73 Incorrect device type in mass storage unit specifer. Check all your device settings and program references.

74 Drive missing or power off.Check your device settings and make sure that the device is receiving power.

75 Disc system error. Possible power difficulties in interface or controller. If necessary, reset the computer.

76 Incorrect unit code in mass storage unit specifier. Check all your device settings and program references.

77 Disc system error.

78 Interface error. Either the wrong type of interface card exists or no interface card exists at the select code specified with the mass storage unit specifier. This error is also generated if the interface card is not both the system controller and the active controller.

79	Reserved for future use.
80	Cartridge out or door open. Also check to see if the interface is connected properly and if the device is ready.
81	Mass storage device failure. Possible power failure. Check the data cable connection or recycle power to the drive.
82	Mass storage device not present. Check mass storage unit specifier. When using flexible discs, check that the unit number is not greater than 3.
83	Write protected. Check the write-protection device on the medium or drive. If it is the result of an INITIALIZE statement to an HP 7906 Disc Drive with the write-protect switch off, then reset the computer (CONTROL STOP)
84	Record not found or disc not formatted properly. A bad spot has been encountered on the medium. You either have to avoid this area of the medium by creating a "dummy" file to avoid attempts to use it, or you have to re-initialize the medium. (Remember, however, that initialization makes all previous data on the medium inaccessible.) When using the 7908 disc drive, this error may also indicate that an attempt was made to address a record outside of the disc address space.
85	Mass storage medium is not initialized or data structure destroyed. Each medium must have been initialized using the INITIALIZE statement. If you are certain that the medium has been initialized, then you have a system failure.
86	Not a compatible mass storage medium. The cartridge or disc must be initialized on a system compatible with your computer.
87	Record address error; information can't be read. Hardware failure. Check for a dirty read head. Perform the System Exerciser if the problem persists.
88	Read data error. Hardware failure. Check for a dirty read head.
89	CHECK READ error. Result of a print verification did not agree with the contents of memory. This occurs only when CHECK READ has been enabled on the file, and after four attempts to write correctly to the file. Check connection on data lines and interface cables.
90	Mass storage system error. Principal cause for this error is "data overrun" (the computer not supplying or receiving data as fast as the disc. Check the disc installation manual for the proper cabling, etc., and retry the operation.
91–99	Reserved for future use.
100	Item in print using list is string but image specifier is numeric.
101	Item in print using list is numeric but image specifier is string.
102	Numeric field specifier wider than printer width.
103	Item in print using list has no corresponding image specifier.
104	ON KBD or TOPEN not allowed in subprogram.
105–109	Reserved for future use.
110	Graphics device specification not recognized. Check spelling of "GRAPHICS", "9872A" or "INCREMENTAL".
111	Graphics device has not been specified. Check select codes.
112	Graphics hardware not installed.
113	LIMIT specifications out of range.
114	HP 98036 Interface Card improperly configured.

115	TDISP not allowed unless peripheral keyboard active.
116	TOPEN is active on another select code.
117–149	Reserved for future use.
150	Improper select code.
151	A negative select code was specified that does not match present bus addressing.
152	Parity error.
153	Either insufficient input data to satisfy enter list, or attempt to ENTER from source into source, or enter count exhausted without linefeed.
154	Integer overflow, or ENTER count greater than 32 767 bytes or 16 383 words.
155	Invalid interface register number. (Can only specify 4-7.)
156	Improper expression type in READIO, WRITEIO, or STATUS list.
157	No linefeed was found to satify / ENTER image specifier, or no linefeed record delimiter was found in 512 characters of input.
158	Improper image specifier or nesting image specifiers more than 4 levels deep.
159	Numeric data was not received for numeric enter list item.
160	Repetition of input character more than 32 768 times.
161	Attempted to create CONVERT table or EOL sequence for source or destination variable which is locally defined in a subprogram.
162	Attempted to delete a nonexistent CONVERT table or EOL sequence.
163	I/O error, such as interface card not present, device timeout, interface or peripheral failure (Interface FLAG line = 0.), STOP key pressed, or improper interface card type.
164	Transfer type specified is incorrect type for interface card.
165	A FHS or DMA transfer with no format specifies a count that exceeds the size of the variable, or an image specifier indicates more characters than will fit in the specified variable.
166	A NOFORMAT FHS or DMA type transfer does not start on an odd numbered character position, such as A$[3].
167	Interface status error, TRL Character or an EOI was received on an HP-IB Interface before ENTER list or image specification was satisfied.
168–183	Reserved for future use.
184	Improper argument for OCTAL or DECIMAL function or assembled location. The OCTAL function has a range from −65 535 to +65 535. The DECIMAL function has a range for its arguments from −177 777B to +177 777B.
185	Break Table overflow.
186	Undefined BASIC label or subprogram name used in IBREAK statement.
187	Attempt to write into protected memory; or, attempt to execute instruction not in ICOM region.
188	Label used in an assembled location not found.
189	Doubly-defined entry point or routine.
190	Missing ICOM statement.

191	Module not found.
192	Errors in assembly.
193	Attempt to move or delete module containing an active interrupt service routine.
194	IDUMP specification too large. Resulting dump would be more than 32 768 elements.
195	Routine not found.
196	Unsatisfied external symbols.
197	Missing COM statement.
198	BASIC's common area does not correspond to assembly module requirements.
199	Insufficient number of items in BASIC COM declarations.
200–206	Reserved for future use.
207	Binaries not allowed in LOAD SUB file. Do LOAD, SAVE, SCRATCH A, GET and STORE on the file to get rid of binaries. However, the loaded program may not run after the binaries are removed.
208	Volume not mounted. Mount it and execute a VOLUME DEVICES ARE statement.
209	Operation not allowed on tape. Only the BKUP file used in DBBACKUP and DBRECOVER is allowed on tape.
210	Bad status array. It must be defined as integer precision with ≥ 10 elements. Check spelling and current size.
211	Improper data base specified or data base not open. Improper name, or performing data base operation with invalid name.
212	Data set not found. Check set name or number and make sure it is on the volume specified in the schema.
213	Reserved for future use.
214	Data base requires creation. Perform a DBCREATE.
215–217	Reserved for future use.
218	Volume name not part of data base. Check spelling.
219	Out of available memory for a DBOPEN, DBBACKUP or DBRECOVER. Out of read/write memory if executed from main program. Out of special area if executed from subprogram, so perform the DBOPEN in the main program.
220	Improper or illegal use of maintenance word. Check spelling and leading or trailing blanks.
221	Data set not created.
222	Reserved for future use.
223	Improper backup file. In DBRECOVER, backup file has incorrect information in header or no primary DBBACKUP/RECOVER currently in progress (for secondary operation).
224	Incomplete backup file. More than one volume in backup; probably mounted in the wrong order. Start the recovery over.
225	Improper utility version number in root file. Rerun Schema Processor to generate new root file.

226	Corrupt data base – must purge and redefine. Purge root file and run Schema Processor.
227	Corrupt data base – all sets require erasure. When erasing a detail data set, ensure that all related master data sets are on-line.
228	Data sets cannot be re-created without root file.
229	Operation not allowed while DBOPEN current. Perform a DBCLOSE mode 1.
230	Improper set list in DBBACKUP, DBCREATE, DBERASE, DBPURGE or duplicate sets in the set list.
231–232	Reserved for future use.
233	Required data set root file not mounted. Mount it and perform a VOLUME DEVICES ARE.
234	Referenced line not a PACKFMT statement. Make sure line identifier is correct and that it references a PACKFMT statement.
235	Reserved for future use.
236	Insufficient length in a PACK statement, or insufficient current length in an UNPACK. Insufficient length in a DBBACKUP or DBRECOVER statement.
237	List length >32 767 in PACK or UNPACK. Array in PACKFMT too large. Make sure it is the correct variable; redimension if necessary.
238	Numeric conversion error. Improper real number found. Check PACKFMT to make sure a REAL or SHORT variable, not INTEGER is being unpacked.
239	UNPACK requires a source string of greater length.
240–299	Reserved for future use.
300	CCOM area not allocated.
301	Not allowed when channel is active.
302	CMODEL statement required.
303	Not allowed when trace is active.
304	Too many characters in CWRITE.
305	New CCOM size not allowed when channel is active.
306	HP 98046 Interface Card failure.
307	Insufficient CCOM allocation.
308	Illegal character in CWRITE of non-TRANSPARENT data.
309	Not allowed for this CMODEL.
310	CCONNECT statement required.
311	Not allowed while Data Comm is suspended.
312	Improper CSTATUS array.
313–329	Reserved for future use.
330	Lexical table size exceeds array size.
331	Improper pointer array*
332	Non-existent dimension specified in MAT REORDER.

Computer Museum

333 Pointer array contains out-of-range subscript value.

334 Pointer array length does not equal number of records.

335 Pointer array is not one-dimensioned.

336 Number of records (plus twice the number of secondary keys plus twice the number of substrings) exceeds 16 383.

337 Subscript extends beyond dimensioned maximum length.

338 Subscript out-of-range in key specifier.

339 Starting location is an out-of-range subscript value.

340 Lexical table is too small to include all characters.

341 Main lexical table length plus mode section length does not equal specified table length.

342 Array is not one-dimensioned or is not integer.

343 Lexical mode section pointer out-of-range.

344 Lexical table length exceeds 16 383.

345 Data type of expression in CASE does not match type of expression in SELECT. Verify that when the SELECT argument is a numeric expression, the CASE argument(s) is a numeric expression. Or verify that when the SELECT argument is a string expression, the CASE argument(s) is a string expression.

346 INDENT parameter out of range. (0 to 72 accepted)

347 Structured construct has improperly matched statements. Verify that you are matching WHILE and END WHILE, REPEAT and UNTIL, LOOP and END LOOP, IF and END IF (only one ELSE is permitted), or SELECT and END SELECT (only one CASE ELSE is permitted). Also verify that the CASE ELSE occurs only after all CASE statements for a given SELECT construct.

348 Attempt to execute looping statement when no matching WHILE, REPEAT, or LOOP is active. Use the INDENT command to determine if you forgot to use a WHILE, REPEAT, or LOOP when writing your program. Use XREF to determine if there are any GOTOs or GOSUBs branching into your structured loops.

* This error occurs when data is lost in the process of reordering the array. If this error does not occur, it does not necessarily imply that the pointer array contains a permutation.

System Error octal number; octal number

This indicates a fatal error detected by the system firmware. It may have been caused by electromagnetic interference, a hardware failure, a firmware error, or an improper command. If reset (CONTROL STOP) does not bring control back, the machine must be turned off, then on again. If the problem persists, contact your local HP Sales and Service Office.

I/O Device Errors

Two error messages can occur when attempting to direct an operation to an I/O device that is not ready for use. A printer which is out of paper or no device at a specifed select code are examples. The first message that appears is –

`I/O ERROR ON SELECT CODE` select code

If the condition is not corrected, the machine beeps intermittently and the following message replaces the first –

`I/O TIMEOUT ON SELECT CODE` select code

The I/O device can be made usable by correcting the error (loading paper, or changing the select code, for example), then executing the READY# command –

`READY#` select code

This command readies the I/O device and the operation which was attempted is attempted again. The select code must be specified by an integer.

If you get an I/O error on select code 0 and the printer is not out of paper, call your Sales and Service Office.

In some cases, such as an interface which is not connected, READY# for that select code may not solve the I/O error. In this case, STOP should be pressed to regain control of the computer. Be sure to turn the power off before inserting an interface. After the problem is remedied, the operation or program can be tried again.

If you get an I/O error and you have an ON KBD statement in effect, you must press STOP to gain control of the computer. Otherwise, the READY# command will be trapped by ON KBD.

CSTATUS Element 0 Errors

10	Timeout before connection
11	Clear to Send line false or missing clock
100	Channel MEMLIMIT overflow
101	Illegal protocol from remote
102	Input buffer overflow
103	Internal buffer overflow
104	Autodisconnect forced
105	RETRIES count exceeded
106	NOACTIVITY timeout
200	98046 buffer overflow

Assembly-Time Errors

Computer Museum

DD	Doubly-defined label
EN	END instruction missing; or module name does not match.
EX	Expression evaluation error.
LT	Literal pools full or out of range.
MO	ICOM region overflow.
RN	Operand out of range.
SQ	Argument declaration pseudo-instruction out of sequence.
TP	Incorrect type of operand used.
UN	Undefined symbol.
900_999	Reserved for user.

IMAGE Status Errors

The following are possible values and meanings of the condition word (first element of the status array). After an error, the status array is as follows –

Element	Description
1	Condition word is non-zero
2-4	No change
5	DBOPEN mode
6	Statement identification number
7	Program line number
8	0
9	Value of the mode parameter
10	Integer-for system use only

Each statement has an identification number.

Number	Statement
401	DBOPEN
402	DBINFO
403	DBCLOSE
404	DBFIND
405	DBGET
406	DBUPDATE
407	DBPUT
408	DBDELETE

Condition Word Value	Error Description
0	Successful execution – no error.
-1	Improper data base name; already have read–write access to the data base.
-10	You may not open additional data bases; five are already opened.
-11	Bad data base name or preceding blanks missing. Don't change the first two characters. Data base may not be open.
-14	DBPUT, DBDELETE and DBUPDATE not allowed in DBOPEN mode 8.
-21	Bad password – grants access to nothing or not to that set. Check spelling. Data item, data set, or volume nonexistent or inaccessible. Check spelling and DBOPEN password. Volume references must be numeric for DBINFO.
-22	Detail data set required.
-23	You lack write access to this data set.
-24	DBPUT or DBUPDATE not allowed on Automatic Master. Check correctness of set reference.
-31	Improper mode in data base statement. DBGET mode 5 bad – specified data set lacks chains.
-52	Item specified is not an accessible key item in the specified set. Bad @ parameter – must be "@;" or "@ " or "@".
-74	Root file name in disc directory and name in root file are different. Make sure root file not moved or renamed.
-91	Root file version not compatible with current IMAGE/45 statements. Incorrect version of Schema Processor used.
-92	Data base requires creation.
-94	Data or structure information lost. Data base must be erased or redefined.
-95	Cannot DBOPEN while a DBBACKUP or DBRECOVER is going on.
11	End of file on serial DBGET; no entries following the current record.
12	Negative record number on directed DBGET. Check record number and spelling.
13	Record number greater than capacity on directed DBGET. Check record number and spelling.
15	End of chain encountered.
16	The data set is full.
17	No current record or the current record is empty; make sure that a current record is defined for this set. There is no chain for the key item value. There is no entry with the specified key value.
18	Broken chain. Must UNLOAD the data base.
41	DBUPDATE will not alter a key item. Make sure correct key item values are in the correct places in the buffer string.
43	Duplicate key item value in master not allowed.

44	Can't delete a Master entry with non-empty detail chains.
50	Buffer string is too small for requested data. Redimension if necessary.
53	Argument parameter type incompatible with key field type (DBGET, mode 7 or DBFIND) or current length of string argument is less than the string length of the key item value.
80	Data set's volume is not on line; or set not created.
94	Corrupt data base successfully opened in mode 8.
1xx	There is no chain head for path xx.
3xx	The automatic master for path xx is full.
4xx	The master data set for path xx is not on-line (Applies to DBPUT and DBDELETE for detail data sets).
500	Root file volume isn't mounted.
5xx	Needed volume on-line; created data set xx isn't there.

Operating and Programming Index

Computer Museum

This subject index is for the following manuals:

I Installation, Operation and Test (09845-93005)
W Workbook (09845-93090)
GW Graphics Workbook (09845-93091)
BP BASIC Programming (09845-93000)
RT Reference Tables
EM Error Messages

a

ABS (absolute value) BP-42
Absolute plotting RT-1
Absolute value (ABS) BP-42
Access rate (tape) BP-206
Accessories supplied I-10
Accessories kits I-10
Accuracy of calculations BP-54
ACS (arccosine) BP-47
Action symbol BP-7
Addition BP-34
Air filters, cleaning I-38
ALPHA GW-4
Alphanumeric keys I-1,I-3,BP-1
Alternate line types GW-8
AND operator BP-35
Angle RT-1
Angular units BP-47
Anisotropic RT-1
Arccosine (ACS) BP-47
Arcsine (ASN) BP-47
Arctangent (ATN) BP-47
Arithmetic:
  Functions BP-42
  Hierarchy BP-48
  Keyboard BP-34
  Operations I-2,W-7,BP-8,BP-34
Array:
  Dimensioning BP-58,BP-59,BP-64
  Functions BP-89
  Identifier BP-58,RT-1
  String BP-95
  Variables . . . BP-13,BP-57,BP-58,BP-76
Arrow keys BP-1
ASCII character codes RT-10
ASN (arcsine) BP-47
ASSIGN W-25,BP-188,BP-196
Assigning a data file . . W-25,BP-188,BP-196
Assigning values to variables I-2,BP-8
Assignment (LET and implied) I-2,BP-56,BP-94
ATN (arctangent) BP-47
Audible output (BEEP) BP-144
AUTO (line numbering) BP-22
Automatic indent BP-20
Automatic start for programs I-5,W-37
AUTOST I-5,W-37
AUTOST key W-2,BP-1
AXES GW-30
Axes RT-1
Axis RT-1

b

BACKSPACE key BP-1
Bar charts GW-22
BASIC language BP-1,RT-19
BASIC User's Club I-36
BEEP BP-144
Binary Coded Decimal Interface exerciser I-83
Binary programs BP-204
Blinking mode BP-218,BP-239
Bounds (of array dimensions) BP-59
Brackets [ ] BP-16,BP-95
Branching BP-114
  End-of-line BP-227
  Looping BP-117
  With SFKs BP-227,BP-230
BUFFER (files) BP-199
Buffering, implicit BP-142
Byte RT-1

C

Calculating range BP-38
Calculations, accuracy of BP-54
CALL BP-136
Calling program BP-129,RT-1
Cartridge Tape Unit exerciser. I-76
Carrying cases, computer I-11
CAT (catalog) I-5,W-17,BP-177
CAT TO BP-179
Characters BP-17,RT-1
  Defining new. BP-252
  Foreign. RT-11
  Nationalized and Drawing RT-11
  Non-printable BP-149
CHECK READ. BP-200
CHECK READ OFF BP-201
CHR$ (character function) BP-106
Circles GW-12
Cleaning:
  Air filters I-38
  Computer I-37
  Light pen I-38
  Tape drives I-37
CLEAR key. BP-1
CLEAR LINE key. BP-1
Clearing the CRT. BP-1
Clearing the keyboard entry area. BP-1
CLIP GW-39,GW-41
Clipping area RT-1
CLR→END key BP-1
Closing a file BP-196
COL (column) BP-89
Color Graphics exerciser. I-45,I-52
Color printing. BP-167
Colored pens GW-8
COM (common). BP-63,RT-15
  In subprograms. BP-138
Comma (for spacing) BP-144
Command. BP-16,RT-1
Comment delimeter (!) BP-9,BP-25
Comments within a program BP-9
Common logarithm BP-46
Compatibility, System 45 . . . . . RT-19
  BASIC RT-19
  File BP-240,RT-21
  Graphics firmware RT-24
  9845A vs. 9845B/C RT-20
  9845A graphics vs. 9845B
    graphics RT-21
  9845B graphics vs. 9845C
    graphics RT-22,RT-24
  9845C graphics vs. enhanced
    graphics RT-22,RT-24
  Standard vs. enhanced processor . .RT-23
Computed GOSUB. BP-123
Computed GOTO BP-114
Concatenation (& - string) BP-97
Connector symbol BP-7
Conserving memory RT-15
Constant BP-16,RT-2
CONT (continue) command. BP-28
  With INPUT. BP-70
CONT key W-4,BP-28,BP-76
Control codes BP-241
  Disabling BP-243
CONTROL
  key W-36,BP-1,BP-23,BP-218,RT-10
Controller address BP-170,RT-2
CONVERGE I-29
Convergence panel I-8
Convergence procedure I-28
Coordinates GW-5
COPY (files). BP-202
Copying an array. BP-78
COS (cosine) BP-47
CREATE (data files) W-24,BP-187
CRT I-8,I-16
  Accessing. BP-146
  Display area I-16
  Installation. I-13
  Intensity control I-8,I-15
  Keyboard entry area. I-16
  Memory BP-239
  Pull-out cards I-8
  Print area. I-16
  Selective addressing. BP-243
  Softkey label area I-16
  Special features BP-218,BP-239
  System comments area I-16
CSIZE GW-14
Current environment. BP-129,RT-2
Current units RT-2
Cursor. RT-2
  Moving. BP-247
  Selective addressing. BP-243

d

DATA (with READ) BP-8,BP-66,BP-98
DATA pointer..... BP-67
Repositioning BP-68
Data (on a mass storage device) BP-187
Amount of storage needed BP-198
Data base RT-2
Debugging BP-210
Decision-making within a program..... BP-8
Decision symbol..... BP-7
DEFAULT OFF BP-51
DEFAULT ON BP-50
Default values BP-50
DEF FN (define functions):
Multiple line BP-128,BP-134
Single line BP-125
Defined record..... BP-173,BP-187,RT-2
Defining a function BP-125,BP-134
Defining special function keys W-14,BP-220
DEG (degrees)..... BP-47
DEGAUSS I-31
DEL (delete line) BP-21
DEL CHR key BP-1
DEL LN key BP-11,BP-21
Deleting characters..... BP-1, BP-248
Delimiter:
Coma..... BP-144
Comment (!) BP-25
PRINT USING BP-156
Semicolon BP-13,BP-144
DET (determinant) BP-90
Device type (mass storage).... BP-171,RT-4
Digit rounding (DROUND) BP-42
DIGITIZE GW-35
Digitizer exerciser I-74
Digitizing..... RT-2
DIM (dimension)..... BP-60
Dimensioning an
array BP-58,BP-59,BP-64,BP-95
Dimensioning a string BP-94
Directory BP-175
DISABLE (interrupts) BP-230
Disc Drive exercisers..... I-68,I-69,I-78
DISP (display) BP-8,BP-144
Display exerciser I-45,I-48
Display keys I-1,I-3,BP-1
Display line..... RT-2
DIV (integer divide)..... BP-34
Division (/) BP-34
By zero..... BP-50
DOT (inner product) BP-90
Dot matrix in syntax BP-16
DRAW..... GW-6
Drawing characters RT-11
DROUND (digit round)..... BP-42
DUMP GRAPHICS GW-4
Dynamic memory allocation..... BP-140

e

e (Napierian) BP-46
E key..... BP-1
EDIT BP-11
EDIT (string)..... BP-99
EDIT KEY (SFKs) BP-220
Edit key mode RT-2
EDIT LINE (programs) BP-19
Edit line mode RT-2
Edit/system command keys..... I-1,BP-1
Editing:
Keyboard lines BP-11
Programs..... BP-19
SFKs BP-220
ENABLE (interrupt) BP-230
END BP-9,BP-29
End of file (EOF) marks BP-174
End of line branching BP-227
End of program (logical and physical). . BP-29
End of record (EOR) marks BP-174
Enhanced Graphics exerciser..... I-45,I-56
Equal to (=) BP-34
Erasing memory..... BP-30
ERRL (error line)..... BP-214
ERRM$ (error message) BP-214
ERRN (error number) BP-214
Error functions..... BP-214
Error messages and warnings.... BP-4,EM-1
Error, system EM-9
Errors: BP-198
Assembly time EM-11
Image status EM-11
I/O device EM-10
Escape code sequences BP-245
Summary..... BP-258
EXECUTE key..... W-4,BP-1
Execution stack RT-17
EXOR operator BP-36
EXP (exponential) function BP-46
Exponential function BP-46
Exponentiation (^ or **)..... BP-34
Expression, numeric BP-17,RT-5

f

Field specifiers BP-155
File:
  Buffer BP-199
  Closing BP-196
  Compatibility BP-240,RT-21
  Name BP-170,RT-2
  Number BP-170,RT-2
  Opening BP-188
  Pointer BP-188,RT-2
  Purging W-18,BP-202
  Specifier BP-172,RT-2
  Structure BP-173
  Table BP-188
File types BP-173,RT-3
  Assembly language BP-178
  Backup BP-178
  Binary data BP-173,BP-178
  Binary program BP-173,BP-178,BP-204
  Data BP-178,BP-181,BP-187
  Data set BP-178
  KEY BP-178,BP-203
  Option ROM BP-178
  Program BP-178,BP-181
  Root BP-178
  STORE ALL (memory) . . BP-178,BP-204
FILL GW-22
Final value (FOR) BP-117
FIXED (fixed point) BP-8,BP-38,BP-39
FLOAT (scientific
  notation) BP-8,BP-38,BP-40
Flowcharts BP-7
Flowline symbol BP-7
FN (function reference) BP-125,BP-133
FN END BP-134
FOR/NEXT BP-8,BP-11,BP-117,BP-121
  Nesting BP-120
Foreign characters RT-11
Formal parameters BP-129,RT-3
Format string BP-155
Formatted output BP-155
FRACT (fractional part) BP-43
FRAME GW-10
Full-precision numbers (REAL) . BP-54,BP-62
Functions BP-8
  Array BP-89
  Defining BP-125,BP-134
  Error BP-214
  In computations W-10
  KBD$ (keyboard) BP-232
  Math BP-42
  Output BP-149
  String BP-103
  User-defined BP-13,BP-125,BP-134
Fuses I-21,I-22

g

GCLEAR GW-10
GDUs GW-18,RT-3
GET BP-182
Glossary RT-1
GOSUB BP-8,BP-122
  Computed BP-123
GOTO BP-8,BP-114
  Computed BP-114
GRAD BP-47
GRAPHICS GW-4
Graphics:
  Display units (GDUs) GW-18,RT-3
  Exerciser I-45,I-52
  Firmware differences RT-24
  Tablet exerciser I-70
  Training program GW-1
Greater than (>) BP-34
Greater than or equal to (≥) BP-34
GRID GW-32
Grounding requirements I-24

h

Handshake RT-3
Hard clip RT-3
Heading suppression (CAT) BP-177
Hierarchy, arithmetic BP-48
HOME key BP-1
Home position (cursor) BP-1
HP-GL RT-3
HP-IB device address BP-18,RT-3
HP-Interface Bus exerciser I-84
HP Compatible BASIC RT-19
HP Sales and Service offices I-93

i

Identifier, array BP-58,RT-1
Identifier, line BP-17,RT-3
Identity matrix BP-82
IF...THEN BP-8,BP-115
IMAGE (with PRINT USING) . . BP-8,BP-155
  Summary BP-166
Implicit dimensioning:
  Array BP-58
  String BP-95
Increment value BP-20
Incremental Plotter Interface exerciser . . I-87
Indent, automatic BP-20
Initial value (FOR) BP-117
INITIALIZE I-4,BP-176
Initializing a tape I-4,W-19
INPUT BP-8,BP-13,BP-69,BP-70,BP-98
Input RT-3
Input/output symbol BP-7
INS CHR key BP-1
INS LN key BP-11
Inserting characters BP-1
Inserting lines BP-11
INT (integer part) BP-43
INTEGER BP-61
Interface:
  Exercisers I-81
  Select codes I-20
  Connecting an I-19
Interleave factor BP-176,RT-3
Interrupt (program) BP-227
  Interaction BP-228
  Simultaneous BP-227
  Types BP-227
Introductory training program W-1
Inverse matrix BP-85
Inverse video mode BP-218
I/O device errors BP-290
I/O slots I-19
Isotropic RT-3

k

KBD$ BP-232
Keyboard I-1,W-4,BP-1
  Arithmetic BP-34
  Entry area I-16,RT-3
  Operations I-2
Keyboard Magazine I-36
Keyword BP-16
  Secondary BP-16

l

LABEL GW-37
Label BP-17,RT-3
LDIR GW-14
LEN (length) BP-103
Less than (<) BP-34
Less than or equal to (≤) BP-34
LET BP-8,BP-54,BP-94,BP-98
  Implied BP-56
LETTER GW-14
LGT (common log) BP-46
Light pen I-8
  Cleaning I-38
  Connector I-21
  Installation I-16
LIMIT GW-39,GW-41
LIN (linefeed) BP-151
Line identifier BP-17,RT-3
Line length BP-18
Line numbers BP-8,BP-9,RT-4
  Auto numbering (AUTO) BP-22
  Range BP-17
  Renumbering (REN) BP-22
  Using EDIT LINE BP-19
Line types, alternate GW-8
LINK BP-184
LINPUT BP-8,BP-98
LIST BP-26
LIST# BP-26
LIST KEY (SFK definitions) BP-225
Literal BP-96
Live keyboard mode BP-3,RT-4
LOAD BP-186
LOAD ALL BP-204
LOAD BIN BP-204
LOAD KEY BP-203
Local variables BP-140,RT-4
LOCATE GW-26,GW-41
LOG (natural log) BP-46
Logarithm:
  Common (LGT) BP-46
  Natural (LOG) BP-46
Logging keyboard operations BP-2
Logical operators BP-35
Logical records BP-173,RT-4
Loop counter BP-11,BP-117
Looping BP-117
LORG GW-37
LWC$ (lowercase) BP-108

m

Main program. BP-17,BP-129,RT-4
Mainframe exercisers I-45
Manuals I-33
  Package I-34
  Structure I-36
Margins. BP-251
Mass storage errors BP-207
MASS STORAGE IS BP-172
Mass storage unit specifier BP-170,RT-4
MAT...CON (constant) BP-76
MAT-copy BP-78
MAT...CSUM (column sum). BP-87
MAT-function. BP-81
MAT...IDN (identity) BP-82
MAT-initialize. BP-77
MAT INPUT BP-71,BP-98
MAT...INV (inverse) BP-87
MAT-multiplication BP-83
MAT-operation BP-80
MAT PRINT BP-153
MAT PRINT # BP-194
MAT READ BP-66,BP-98
MAT READ # BP-194
MAT...RSUM (row sum) BP-88
MAT-scalar operation BP-99
MAT...TRN (transpose). BP-87
MAT...ZER (zero) BP-76
Matrices BP-82
  Identity. BP-82
  Inverse BP-85
  Multiplication. BP-83
  Transpose of BP-87
MAX (maximum) BP-43
Maximizing performance. RT-23
Medium. RT-4
Memory RT-14
  Allocation, dynamic BP-140
  Available for use BP-26
  Conserving RT-15
  DMA and FHS NOFORMAT
    transfers RT-18
  Erasing. BP-30
  Exerciser I-45,I-47
  Lock. BP-249
  Loss RT-14
  Organization RT-14,RT-15,RT-17
  Read/Write. I-2,I-9,RT-6,RT-14
  Storing. BP-204
  Test I-26,RT-14
  Types of. RT-14
  Working storage RT-14
Metric conversion table. RT-12
Metric units. RT-4
MIN (minimum) BP-43
Minus sign (–). BP-34
Mnemonic. RT-5
MOD (modulo) BP-37
Module RT-5
MOVE GW-6
MSCALE. GW-18
Msus BP-170,RT-5
Multiple-line function
  subprogram BP-128,BP-134
Multiplication (*) BP-34

n

Name. BP-17,BP-55,RT-5
Naming convention RT-5
Napierian e. BP-46
Nationalized and Drawing characters. . RT-11
Natural logarithm (LOG). BP-46
Nested FOR/NEXT loops BP-120
NEXT BP-117
Non-printable characters BP-149
NORMAL BP-213
Not equal to (< > or #) BP-34
NOT operator BP-36
Null string. BP-103
NUM (numeric) BP-107
Number formats for output. BP-38
Numeric:
  Expression BP-17,RT-5
  Keys I-1,I-2,BP-1
  Variable BP-54

o

OFF END BP-198
OFF ERROR BP-215
OFF KBD BP-234
OFF KEY BP-237
Offices, HP Sales and Service I-93
ON END. BP-197
ON ERROR BP-213
ON...GOSUB. BP-123
ON...GOTO BP-114
ON INT. BP-227
ON KBD. BP-227,BP-230
ON KEY BP-227,BP-234
On-line RT-5

Opening a file BP-188
Operating procedures I-1
Operators BP-34
  Arithmetic BP-34
  Logical BP-35
  Relational BP-34
  String BP-97
OPTION BASE BP-60
OR operator BP-36
Origin RT-5
Output BP-143,RT-5
Output functions BP-149
Output of numbers BP-38
Outputting program results BP-8
OVERLAP BP-167
Overlaying RT-15

p

PAGE BP-152
PAPER ADVANCE key I-27
Paper, internal printer I-11,I-27
  Installation I-27
Paper Tape Punch exerciser I-78
Paper Tape Reader exerciser I-78
Parameters BP-129
Parentheses BP-49
Parity RT-5
Pass by reference BP-131
Pass by value BP-131
Pass parameter BP-129,RT-5
PAUSE BP-28
PDIR GW-12
Pen RT-5
PENUP GW-6
Performance, maximizing RT-23
Peripheral exercisers I-67
Peripherals I-32
Physical records BP-173,RT-5
PI BP-44
Pie charts GW-12,GW-25
Pixel RT-5
PLOT GW-6
Plotted point RT-5
Plotter exerciser I-73
PLOTTER IS "GRAPHICS" GW-10
Plotting:
  Absolute RT-1
  Coordinates RT-6
  Space RT-6
  Plus sign (+) BP-34

Computer Museum

  Pointer RT-6
  DATA BP-67
  File BP-188
  Program BP-18
  Repositioning (file) BP-194
POLYGON GW-12,GW-24
POS (position) BP-104
Power:
  Cord I-24
  Cord connector I-21
  Requirements I-23
  Switch I-8,I-25
Power-of-ten rounding (PROUND) . . . BP-44
Precision (accuracy) BP-54
  For calculating BP-54
  For conserving memory RT-15
Pre-run initialization BP-27
PRINT BP-8,BP-147
PRINT # BP-189,BP-193
PRINT ALL IS BP-2
PRINT USING BP-8,BP-155
Print all mode W-29,BP-2
PRINTER IS I-4,BP-8,BP-9,BP-146
Printer:
  Addressing BP-250
  Exercisers I-45,I-51,I-70,I-71,I-76
  Internal I-8,I-27,BP-146
  Paper I-11,I-27
  Margins BP-251
  Select Code I-4,I-20,W-9,BP-2
  Standard RT-5
Printout area, CRT RT-6
Priority BP-227,RT-6
Processors, standard vs. enhanced . . . RT-23
Program BP-18
  Control keys I-1
  Editing BP-9
  Outline BP-6
  Pointer BP-18
  Running BP-27
  Segment BP-17,BP-129,RT-6
  Writing I-3,BP-8
Program flow, controlling BP-8
Programming tutorial BP-5
Prompt BP-69
PROTECT BP-201
Protect code BP-172,BP-201,RT-6
PROUND (power-of-ten round) BP-44
PRT ALL key BP-1,BP-2
PURGE BP-202
Purging files W-18,BP-202

r

Random file access BP-193
Random number (RND) BP-44
Random number seed. BP-44
  Scrambling BP-45
RANDOMIZE BP-45
Range:
  Calculating. BP-38
  Line numbers BP-17
  Various variable precisions BP-54
  Storage BP-38
RAD (radian) BP-47
READ (with DATA) BP-8,BP-66,BP-98
READ # BP-191,BP-194
Read Only Memory (ROM). RT-6,RT-14
Read/Write Memory. I-2,I-9,RT-6,RT-14
Reading data from a file W-27
READY # EM-10
REAL BP-54,BP-62
Real Time Clock Interface exerciser I-85
Recall buffer. BP-1
RECALL key BP-1
Record I/O RT-6
Recording data in a file. W-26
Records BP-173
  Defined BP-173,BP-187,RT-2
  Logical BP-173,RT-4
  Physical BP-173,RT-5
RECTANGLE. GW-22
REDIM (redimension) BP-64
Redim subscripts BP-64,RT-6
Redimensioning an array BP-64
Reference tables. RT-1
Reflected plot RT-6
Relational operators BP-34
Relative plotting. RT-6
REM (remark) BP-25
Remarks in program lines BP-9,BP-25
REN (renumber) BP-22
Renumbering lines. BP-22
RENAME (file) BP-203
REPEAT key. BP-1
RE-SAVE BP-184
Reset. W-35,BP-2,BP-14,BP-29,RT-13
RE-STORE. BP-186
RESTORE (with READ, DATA) BP-68
RES (result) function I-2,BP-1,BP-44
Result buffer. BP-34
RESULT key. I-2,BP-1,BP-34
RESUME INTERACTIVE BP-4
RETURN:
  With DEF FN. BP-134
  With GOSUB BP-122
Return variable BP-188
REV$ (reverse) BP-108
REWIND. BP-207
RND (random number). BP-44
ROLL keys BP-1
Rolling the display. BP-248
ROM RT-6,RT-14
  Drawers I-8,I-17
  Installation. I-18
ROM Revision exerciser I-45,I-65
Roman Extension Character Codes. . . RT-11
Rounding BP-41
  Digit (DROUND). BP-42
  Power-of-ten (PROUND). BP-44
ROW. BP-89
RPT$ (repeat) BP-108
RS-232 Interface exerciser I-86
RUN BP-27
RUN key W-4
RUN light I-8,BP-18
Run-time error trapping BP-125

s

Sales and Service offices. I-93
SAVE BP-182
SCALE GW-18,GW-41
Scalar RT-7
Scientific notation (FLOAT) BP-38
SCRATCH BP-30
SCRATCH A BP-30
SCRATCH C BP-30
SCRATCH KEY. BP-30,BP-225
SCRATCH P BP-30
SCRATCH V BP-30
Secondary keyword. BP-16
SECURE (program lines) BP-30
Select codes I-20,BP-18,BP-170,RT-7
Selective addressing (CRT) BP-243
Selective catalog specifier. BP-177
Semicolon (for spacing) BP-13,BP-144
SERIAL (mode). BP-167
Serial file access. BP-189
SFKs I-1,W-14,BP-1,BP-217,RT-7
SGN (sign) BP-45
SHIFT key BP-1
SHORT. BP-62
SHOW GW-18
Significant digits. BP-41
  In computations BP-54
Simple variables. BP-54
Simultaneous computations I-2
Simultaneous interrupts BP-227

SIN (sine) BP-47
Sixteen-Bit I/O Interface exerciser I-82
Slant RT-7
Snapshot. RT-7
Soft clip. RT-7
Softkeys I-8
SPA (space) BP-150
Space dependent mode W-28,BP-23
Spacing between characters I-2,BP-23
Spare directory BP-175
Special function keys
(SFKs). I-1,W-14,BP-1,BP-217,RT-7
Defining as typing aids. . . . W-14,BP-220
Erasing definitions BP-225
Listing definitions BP-225
Pre-defined definitions. BP-218
Program interrupts BP-227,BP-234
Special features. BP-218
Split plots GW-26
SQR (square root) BP-45
Stack. RT-7
STANDARD output format BP-8,BP-39
Standard mass storage device . . RT-7,BP-172
Standard printer. RT-7
Statements BP-8,BP-16,RT-7
STEP key. W-30,BP-27
Stepping through a program BP-27
STOP BP-29
STOP key. BP-28
Storage of variables:
In memory BP-38,BP-73,RT-18
On mass storage devices BP-199
Storage range. BP-38
STORE. BP-185
STORE ALL. BP-204
STORE BIN BP-204
STORE KEY BP-203
STORE key I-3,BP-18
Storing a program line. I-4
Storing a program on tape I-4,BP-13
String:
Array BP-95
Expression BP-96,RT-8
Functions. BP-103
Operators. BP-97
Variable BP-13
Strings. BP-93
Comparing BP-110
Concatenation. BP-97
Dimensioning (explicit). BP-94
Dimensioning (implicit) BP-95
Maximum size BP-94
Relational operations BP-110
SUB BP-136
SUBEND BP-136
SUBEXIT BP-136
Subprograms . . BP-17,BP-128,BP-138,RT-8
Conserving memory with. RT-15
Function subprograms . . BP-128,BP-134
Subroutine subprograms
(SUB) BP-128,BP-136
Subroutine return pointers BP-122
Subroutines (GOSUB) BP-122,RT-8
Subscripts BP-59,RT-8
Substring BP-96
Substring specifier BP-96
Subtraction (−) BP-34
SUM BP-89
SUSPEND INTERACTIVE BP-3
Symbol table RT-17
Symbols, flowchart BP-7
Syntax conventions. BP-16
System comments line. RT-8
System error EM-9
System exerciser cartridges I-41
System exerciser summary table I-91

t

TAB (output function). BP-11,BP-149
TAB key BP-1
TAB CLR key. BP-1
TAB SET key BP-1
Tab capabilities BP-1
Using escape codes BP-247,BP-251
Table, attaching computer to a I-12
TAN (tangent) BP-47
Tape cartridge:
Access rate BP-206
Blank I-11
Capacity BP-206
Care. I-39
Catalog I-5,W-17
Inserting and removing I-31
Length of. BP-206
Optimizing use BP-207
Specifications BP-206
Unprotected I-3,W-1,BP-205
Write-enabled. I-3,W-1,BP-205
Write protection I-3,W-1,BP-205
Tape drives. I-8
Cleaning I-37
Tape exerciser I-45,I-50
Terminator symbol BP-7

Testing the computer I-41
Text BP-17,RT-8
Thermal printer, internal I-8,I-27
TOP OF FORM key I-27
TRACE BP-210
TRACE ALL BP-213
TRACE ALL VARIABLES BP-212
TRACE PAUSE BP-211
TRACE VARIABLES BP-212
TRACE WAIT BP-211
Tracing variables W-33
Tracing program execution W-33
Training tapes I-32,W-3,GW-3
Transpose of a matrix BP-87
Trigonometric functions BP-46
TRIM$ BP-108
Truth table BP-37
TYP (data type) BP-196
TYPEWRITER OFF BP-31
TYPEWRITER ON BP-31
Typewriter keys I-1,BP-1
Typewriter mode BP-1
TYPWTR key BP-1

u

UDUs (User-Defined Units) GW-18,RT-8
UNCLIP GW-39
Unconditional branching BP-114
Underline mode BP-218,BP-231
Unit code BP-171,RT-8
  9885 . . . . . BP-171
Unpacking the computer I-9
UPC$ (uppercase) BP-107
User-defined function BP-13,BP-125,BP-134
User-defined units (UDUs) GW-18, RT-8

v

VAL (value) function BP-105
VAL$ BP-105
Value area RT-17
Variables I-2,BP-9,BP-13,BP-53,RT-8
  Array BP-13,BP-57,BP-58,BP-76
  Assigning values to BP-56,BP-65
  Breakdown BP-55,BP-111
  Forms BP-54
  Local BP-140,RT-4
  Names BP-55
  Numeric BP-54
  Precision BP-54
  Ranges BP-54
  Return BP-188
  Simple BP-54
  Storage BP-38,BP-73,RT-18
  Types BP-54
Vectors BP-57
Verification (file) BP-200
Vertical line (in syntax) BP-16
Voltage selector switch I-21,I-22

w

WAIT BP-31
WIDTH BP-146
Word RT-9
Working storage RT-14
Write protection (tape
  cartridge) I-3,W-1,BP-205