Hewlett-Packard System 45 Desktop Computer

Mass Storage Techniques

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Mass Storage Techniques

HP System 45 Desktop Computer

Hewlett-Packard Calculator Products Division
P.O. Box 301, Loveland, Colorado 80537
(For World-wide Sales and Service Offices see back of manual.)

Manual Changes

System 45
Desk Top Computer
Mass Storage Techniques
For Manual Data Sept. 1, 1977

page 13:

The second sentence of the next to last paragraph should read –

With a flexible disk, each physical record has a write-check performed upon it and tracks with any defective records are rejected and ignored in future processing.

Error Messages Sticker

Along with the Mass Storage ROM, you received a self-adhesive label (P/N 7120-6835) containing the additional error messages used by the ROM. This label should be placed on the pull-out card underneath the CRT provided for the purpose. This will provide you with a future quick reference for the error messages. A full discussion of the errors and error processing can be found in Chapter 6.

page 16:

Add a new section to end of the page to read –

Flexible Disk Master-Slave System

Whenever using a flexible disk master-slave system, be sure that all units on the same select code are turned on. All must be turned on in order for **any** of the units to be accessed with the statements and commands discussed in this manual.

page 18:

In the second paragraph of the "Records" section, change as follows –

A defined record may be any size between 4 and 32,767 bytes...

page 20:

In the last paragraph on the page, chang: s follows –

Where each record is physically located is highly dependent upon the device itself.

page 30:

In the first paragraph on the page, change as follows –

The maximum length is 32,768 bytes.

page 33:

Add the following note before the "Writing Records" section –

CAUTION

WHEN USING A DISK SYSTEM WITH MORE THAN ONE UNIT ON A SELECT CODE, DO NOT RE-ASSIGN A FILE NUMBER FROM ONE UNIT TO ANOTHER USING THE SAME SELECT CODE UNLESS YOU FIRST CLOSE THE FILE. RE-ASSIGNING A FILE WITHOUT CLOSING IT FIRST IN SUCH CIRCUMSTANCES WILL CAUSE A SYSTEM DEADLOCK.

After the second paragraph, insert a new paragraph –

ASSIGN is always performed in the SERIAL (non-overlapped) processing mode, regardless of whether overlapped processing is currently in effect. Overlapped processing for other statements is not affected by the presence of an ASSIGN statement.

page 34:

In the second paragraph of the "Print Verification" section, change the last sentence as follows –

If they are not identical (implying there has been sort of failure, either on the write or the read-after-write operation), it will try writing and verifying the physical record three more times before giving you an error message indicating that you have a problem.

page 35:

In the note, change –

...it is recommended that CHECK READ be used only when data correctness is of paramount concern.

page 42:

Replace the cautionary note with the following paragraph –

FPRINT and FREAD are always performed in the SERIAL (non-overlapped) processing mode, regardless of whether overlapped processing is currently in effect. Overlapped processing for other statements is not affected by the presence of FPRINT or FREAD statements.

page 53:

Change the first line of the example to read –

```
PROTECT  "Employ:T15","Pay"
```

page 61:

Change line 500 of the example to read –

```
500  PRINT #1;A(*)
```

page 65:

Change line 1020 of the example to read –

```
1020 DISP "Please insert backup floppy"
```

page 67:

Change the second paragraph to read –

The ASSIGN, FPRINT, and FREAD statements are are always executed in a non-overlapped (SERIAL) mode, regardless of the OVERLAP status. The OVERLAP mode also cancels any ON ERROR trapping for CHECK READ.

page 78:

The READY command should read –

READY #nn

page 83:

The description of ERROR 61 should be as follows –

Defined-record size is too small for the data item. Either expand the defined-record size, or try to fit the data into more than one record.

page 90:

Add to the description of inserting the tape cartridge –

If the latch will not engage, press on the (eject) bar above the unit. The cartridge should then be able to be inserted and latched.

page 92:

Add to the last paragraph of the "Rewinding the Tape" section –

But be careful in using this method – reset will halt operations on *every* device connected to the system, not just the cartridge concerned!

page 93:

Add the following note at the end of the page--

NOTE

Occasionally during the use of a tape cartridge, unexpected high-speed tape movement may occur. This action is to assure proper tape tension and does not affect processing or programs.

---

page 94:

To the last paragraph on the page, add the following –

After this file is created, however, it is not possible to create files of more than 426 physical records until the file is purged.

page 104:

The first listed syntax for the ASSIGN statement should not contain an asterisk.

The following are additional allowable syntaxes for the ASSIGN statement –

ASSIGN # file number TO *[, return variable [, protect code]]
ASSIGN * TO # file number [, return variable [, protect code]]

page 105:

The FPRINT syntax should read –

FPRINT file specifier [, protect code] , array identifier

and the FREAD syntax should be –

FREAD file specifier [, protect code] , array identifier

page 107:

In the RENAME syntax –

old file specifier, **new file name** and **protect code** are string expressions.

Table of Contents

Chapter 1: General Information

Overview . . . 1
- Uses for Mass Storage . . . 2
- Buzzwords . . . 4

Fundamental Syntax . . . 5
- Syntax Conventions . . . 5
- Mass Storage Unit Specifier . . . 5
- MASS STORAGE IS Statement . . . 8
- File Names . . . 8

Chapter 2: Getting Started

Initial Steps . . . 11
- Mass Storage ROM . . . 11
- Initialization . . . 13
- Interleaving . . . 14
- Flexible Disk Master-Slave System . . . 16
- Data Compatibility with Other Systems . . . 17

Storage Operations . . . 18
- Types and Methods of Storage . . . 18
- Records . . . 18
- Files . . . 20
- File Directory . . . 21
- Types of Files . . . 23
- Types of Access . . . 24
  - Serial Access . . . 25
  - Random Access . . . 28
- Creating Files . . . 29
- Record I/O . . . 31
- Writing Records . . . 33
- Print Verification . . . 34
- Reading Records . . . 36
- Using Serial and Random Access Together . . . 39
- Rapid Transfer of Arrays . . . 39
- Previewing a Data Item . . . 42
- User-Controlled End-of-File . . . 45

Chapter 3: Storage Management

Fundamentals 47
Selecting Record Size 47
Overflowing Files 48
Copying Files 50
Purging Files 50
Special Operations 52
Protecting a File 52
Renaming Files 53
Execution from the Keyboard 53

Chapter 4: Data Transfers

Buffering 55
Device Buffering 55
Overriding the Device Buffer 57
Device Buffer Memory Requirements 58
Additional Buffering 59
Conflict Between CHECK READ and BUFFER 61
Using Arrays as Buffers 61
Advanced Techniques 63
Passing Data Between Programs 63
Backup Files 65
Overlapped I/O 66

Chapter 5: Non-Data Files

Normal Usages 69
Storing Programs 69
Storing Key Definitions 74
Special Situations 75
Loading Binary Programs 75
Memory Snapshots 75
Effect of CHECK READ 76

Chapter 6: Errors and Error Processing

Hardware Errors 77
Hardware-Related Errors 77
What To Do About Hardware Errors 79
Anticipating Hardware Errors 79
Error Messages (Hardware) 80
Software Errors 82
Software-Related Errors 82
Anticipating Software Errors 85

Appendix A: **ASCII Character Set** 87
Appendix B: **Internal Tape Cartridge** 89
General Physical Information 90
Inserting the Tape Cartridge 90
Removing the Tape Cartridge 91
Write-Protection 91
Rewinding the Tape 92
Tape Care 92
Environmental Considerations 93
Tape Structure 94
Appendix C: **Disk Drives** 95
Interfacing 95
Available Devices 95
Disk System Test 96
Timings 96
Appendix D: **Mass Storage Command Basic Syntax** 103
Appendix E: **Error Messages** 109
Mass Storage ROM Errors 112
Graphics ROM Errors 112
Appendix F: **Maintenance** 115
Maintenance Agreements 115
HP Sales and Service Offices 116
Index 118

Chapter 1

General Information

Overview

This manual is intended for use by HP 9800 Series System 45 Desktop Computer users who are installing or using mass storage devices with their desktop computer. Discussion in this manual will focus upon effective use of the mass storage system, relying upon the Unified Mass Storage Concept employed by the System 45.

The objectives for the manual are –

- To provide all necessary information to allow you to utilize compatible mass storage devices with the System 45.
- To outline effective techniques involving the Unified Mass Storage Concept.
- To isolate potential trouble areas involving mass storage devices and commands, and to provide approaches for dealing with the problems arising from those areas.

To meet these objectives, this manual has been organized around an "objectives-oriented" approach, as opposed to a strict syntactical or semantical treatment. Consequently, you may find this difficult to use as a "quick reference" for syntax and meaning for many of the mass storage commands. Appendix D, Basic Mass Storage Command Syntax, has been provided to meet this objection, though it is recommended that quick reference be made primarily to the operating and programming manual or reference guide provided with the System 45.

It is assumed throughout the treatment herein that you are familiar with the basic operation and language of the System 45. It is not necessary, however, that you be familiar with any aspect of mass storage operation or programming.

The Unified Mass Storage Concept is an approach which enables you, as a programmer, to rely upon the device-independence of mass storage statements used in your programs. It is designed so that writing a record to a disk, for example, will be in all possible respects the same as writing a record to a tape. The concept should enable you to be able to switch your application from one type of device to another, with a minimum of disruption to your program's logic.

Of course, there are still differences between devices which will have an impact upon your programming. These will be pointed out to you. Where there are differences between whole classes of devices – such as between tapes, flexible disks, and hard disks – they will be included in the body of the text.

Particular operating, installation, and maintenance information for HP mass storage peripherals can be found in the operating manual for that device. Even though you are familiar with the material in the body of this text, if you have recently acquired a mass storage peripheral, it is advisable to consult Appendix C, and the device's operating manual.

Throughout this manual, you will find "disk" as the preferred spelling over "disc".

In addition to the statements discussed here, there are two additional programming statements provided by the Mass Storage ROM – GSTORE and GLOAD. These are statements usable only with the graphics system provided by the Graphics ROM (HP product number 98437A). For information on the uses of these statements, consult the CRT/Graphics Operating and Programming Manual (part number 09845-90050).

We at HP are constantly trying to improve our product and produce manuals which are of value to you. To help us meet your needs, we would appreciate your taking some time to complete the post-paid questionnaire at the end of this manual and returning it to us.

Uses for Mass Storage

Mass storage is primarily a means of storing information. Most storage activities and operations center upon this function. Working with mass storage is normally required in applications which assume the retention of information in machine-readable form so that it might be used at a later date, or where there is need for a repository of information which exceeds the internal memory capabilities of the System 45, or both.

Mass storage devices and media were developed for both of these reasons. Most computers, the System 45 included, do not have the capability of retaining information in their memories once the power to the unit has been shut off. Since most people don't leave the machine on constantly, but still have information they would like to have available from turn-on to turn-on, a means of saving that information becomes desirable. Mass storage is such a method of saving information – be it program, data, or even the entire machine state – so that it can be retrieved later and used by the machine again.

This capability also applies to those who might be interrupted by another user of the machine. Rather than having to make a person wait until you are finished, and instead of having to reconstruct what you were doing when you get "bumped" by someone with a higher priority, mass storage can be employed to save information and permit you to return later after they are done. Still another use is the situation where you have programs or data which will be used at a future time – perhaps frequently.

Another inherent problem with every computer is the fact that its memory resources are not infinite. It is quite possible – easy, in fact, with a machine of any size – to exhaust the entire memory of the machine with a program and data and still have a need for more. In such circumstances, mass storage devices can come to the rescue offering a capability of storing large amounts of information in an easily accessible form. A program could then access this data as it requires it, instead of keeping it around gobbling up memory when it isn't being used.

These are the primary reasons for considering mass storage in an application you might have in mind. If any one of these considerations, in some form or another, happens to pop up, you probably have a mass storage application. Some of the most common variations of these are –

- Saving a program.
- Saving the special function keys.
- Retaining the entire memory state of the machine.
- Storing parts of programs which, in their entirety, are too large to fit into memory.
- Creating data with one program to be used by another.
- Making provision for recovery in case something unpleasant and unexpected happens.
- Keeping activity logs and data from real-time acquisitions.

Buzzwords

During the course of the discussion in this manual, phrases will be used which are in common circulation in the data processing industry. While the meaning of most are either well-known or deducible from the context, there are a few which may be new to the user not exposed to mass storage before –

byte – a group of 8 binary digits (bits) operated upon as a unit.

data base – a set of data which is accessible by the computer and upon which a program may perform operations.

file pointer – the current position within a file where data is about to be read or written.

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 disk packs are examples of "media".

mnemonic – an abbreviation or acronym that is easy to remember.

module – in programming, a program segment which performs a specific, independent program task.

naming convention – a pattern or system for assigning names to variables or files so that some manner of consistency or predictability is maintained.

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.

record I/O – input/output operations concerned exclusively with the smallest addressable unit of storage (records).

snapshot – current state at a particular time.

stack – a portion of memory used to temporarily hold information for processing in a particular order.

system design – the specification and implementation of a program or set of programs to accomplish a given purpose.

Fundamental Syntax

Syntax Conventions

The syntax conventions used in this manual are those used in the Operating and Programming Manual –

dot matrix	All items displayed in dot matrix should appear as shown.
|	Items separated by a vertical bar means that one item *or* the other may be used.
/	Items separated by a slash means that either item, or both, may be used.
[ ]	Items contained in brackets are optional items.

Mass Storage Unit Specifier

Many commands use what is known as a "mass storage unit specifier", or **msus**. This specifier tells the System 45 what type of peripheral it is addressing and where it can be "found". The **msus** is a *string* which looks like this –

: device type [, select code [, controller address | 9885 unit code [, unit code]]]

The device type is a capital letter designating the type of peripheral. The permissible codes are –

Code	Device
C	Removable disk cartridge (HP7906)
D	Fixed disk (HP7906)
F	Flexible disk (HP9885)
P	Removable disk pack (HP7920)
T	Tape cartridge (internal to the computer)
Y	Removable disk cartridge (HP7905A)
Z	Hard disk (HP7905A)

The **select code** is an integer in the range of 1 through 12, 14, and 15. For the *standard* tape cartridge unit (the one on the right-hand side), this code is 15. For the *optional* tape cartridge unit (the one on the left-hand side), it is 14. These two codes are permanently reserved for these units. Three other codes are also reserved – 0, for the keyboard (and optional printer); 13, for optional graphics; and 16 for the CRT. Thus, permissible select codes for external mass storage devices are 1 through 12. This select code is the setting on the interface for the device (consult Appendix C for the location of interfacing information for your particular device). Do not choose a setting which is already in use by another device.

The **controller address** is the device address of the controller, if you are using a master-slave system on the select code. It is used only for disk systems in the HP 79-series of drives. The address may be any integer from 0 through 7. Consult the operating manual of the controller involved for the location of the device address switch. If there is only one controller on the select code, the **controller address** may be omitted. If omitted, the address defaults to 0, and the device address switch on the controller must also be set to 0.

If you are using a flexible disk master-slave system, the **9885 unit code** is used in place of the **controller address**. It is the device address of the unit being referenced. The address may be any integer from 0 through 3. Consult the Installation Manual for the HP 9885 for the location of the drive-selection switch.

The **unit code** is the drive address (or drive number) associated with a particular drive in a master-slave system on the select code. If omitted, the code defaults to 0, and the address switch on the drive must also be set to 0. It is not used with flexible disks and is ignored if included with an F device type.

Some examples of mass storage unit specifiers –

If you want to specify the standard tape unit –

```
:T15
```

If you want to specify the optional tape unit –

```
:T14
```

If you want to specify a flexible disk drive with an interface setting (select code) of 8, controller 0 –

```
:F8,0
```

If you want to specify the HP7905's removable disk cartridge, on a master-slave system, select code 4, controller 0, drive 3 –

```
:Y4,0,3
```

If you want to specify an HP7905 hard disk on select code 9, controller 0, drive 0 –

```
:Z9,0
```

If you want to specify a removable disk pack on select code 1, controller 0, drive 0 –

```
:P1
```

There are certain *default* select codes, and they are implied if omitted. They are –

```
:F  implies :F8,0
:P  implies :P12,0,0
:T  implies :T15
:Y  implies :Y12,0,0
:Z  implies :Z12,0,0
```

It was said that the **msus** is a string. Actually, it may be formed by any string *expression* which creates a valid specifier. For example, if the following statements have been previously executed –

```
A$ = ":T15"
Type$ = "T"
Select = 15
```

then the following would all be valid specifiers (in this case, all meaning ":T15") –

```
":T15"
A$
":" & Type$ & VAL$(Select)
A$[1;2] & "15"
```

Expressions may also represent the default forms, so –

```
A$[1;2]
":" & Type$
```

would also represent the standard tape unit.

In the rest of the manual, you will see constant references to the **msus**. This notation throughout will denote that any of the above forms may be used.

MASS STORAGE IS Statement

Except for the INITIALIZE statement which is treated in Chapter 2, in all references below, the **msus** may be omitted and the *default* mass storage device will be assumed. The default device is ordinarily :T15 (this is the power-on value). It may be changed to any other device by executing –

```
MASS STORAGE IS msus
```

After executing this statement, all future defaulted **msus** references will automatically assume the device indicated by this statement. The statement may be executed as many times as desired, and may be executed from the keyboard or from a program.

A MASS STORAGE IS statement can be overridden and the standard **msus** (:T15) restored by any of the following –

- Power-off, then power-on.
- Executing SCRATCH A.
- Executing MASS STORAGE IS ":T15" (or ":T").

File Names

You will encounter many references throughout this manual to a "file" name. The concept of a file, and the manipulation of it, will come later in the manual. However, it can be said in advance that all files stored on a mass storage must have a *name*.

A file name is a string, just like an **msus**. It may be anywhere from 1 to 6 characters long and may contain any character except –

NULL	ASCII decimal value 0
Quote-mark (")	ASCII decimal value 34
Colon (:)	ASCII decimal value 58
(unnamed)	ASCII decimal value 255

These names must be unique on a given medium, but files bearing the same name can be stored on *different* media. Uppercase letters are different characters than lowercase, thus the file name "GEORGE" is different from "george", which is different still from "George" and "geORge", and so on. Some examples of file names –

Test
TEMP
FILE2
Backup
-KEEP-
$$$$$$

Blanks (ASCII decimal value 32) in a file name, whether leading, trailing, or imbedded, are ignored. Thus " A B " as a file name would be treated as "AB". Non-printing characters may be included, and comprise part of the character-count for the string, but their existence may not be noticeable in any listing of the file name.

Since a file name is a string, as with the **msus**, any string *expression* may be used to create it. Thus –

```
File$ = "TEMP" & VAL$(Counter)
```

would create a string which, as long as the length of File$ was less than (or equal to) 6, could be used as a file name where a file name is required. For example, if Counter were 3, using File$ would be referencing a file called "TEMP3".

Attempts to create or access a file with a file name greater than six characters (ignoring blanks), or containing any of the prohibited characters, will result in an "improper file name" error (error number 53).

File Specifier

A "file specifier" is defined as a file name, or a file name followed by an **msus** –

file name [msus]

The inclusion of an **msus** is used to direct particular files to (or from) selected devices which are on line at a given time. Thus, if you wanted to reference a file called "Backup" on a flexible disk with select code 10, you could say –

Backup:F10

or the same file on the standard tape cartridge unit –

Backup:T

Since both the file name and the **msus** are strings, they may be stored separately and concatenated together when needed. For example –

```
File$ = "Volts"
Device$ = ":F"
```

and hence

File$ & Device$

would produce a reference to

Volts:F

i.e., the "Volts" file on the flexible disk (select code defaulted to 8).

NOTE

When a file specifier is referenced without an **msus**, the System 45 defaults the reference to the standard mass storage device as designated by the last MASS STORAGE IS statement (if one has been executed since power-on, or since the most recent SCRATCH A command), or to the power-on default device (the standard tape cartridge unit).

Wherever reference to a file specifier is required, with or without an **msus**, the notation **file specifier** will be used.

Chapter 2

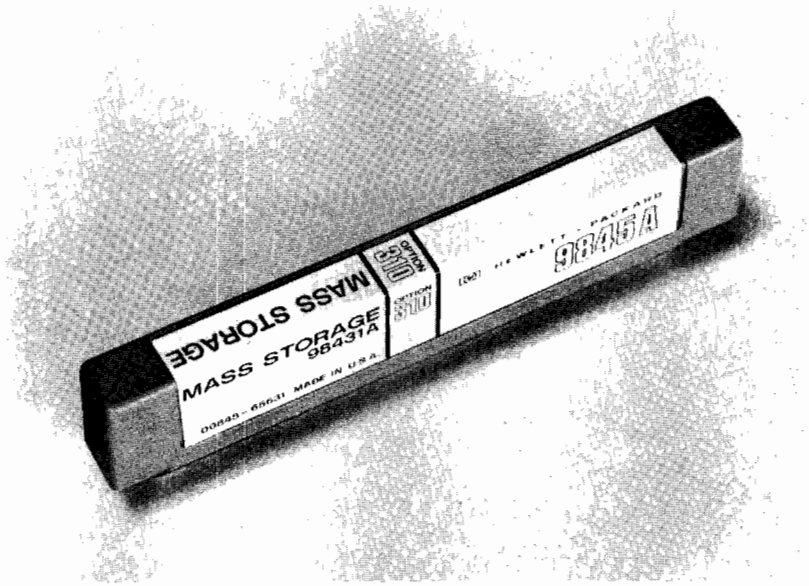
Getting Started

Initial Steps

Mass Storage ROM

With the exception of the internal tape cartridge units, no mass storage device can be used unless the HP98431A Mass Storage ROM has been installed in the computer (Option 310 if factory-installed). The ROM is very simple to install.

There are two ROM drawers for the computer, one on each side of the machine. There are two types of ROMs in each drawer – the operating system ROMs (which must always be in place), and the option ROMs (one or more of which may be present). The two types are color-coded, with the operating system ROMs being brown plastic and all option ROMs being yellow plastic. The Mass Storage ROM has a red label, indicating it goes into the ROM drawer on the left.



The Mass Storage ROM

To add the Mass Storage ROM, turn off the computer, slide the ROM drawer on the left side of the machine all the way out, and open the clear plastic cover. Orient the ROM so that the label reads the same way as the others in that drawer and insert it vertically in one of the unused option ROM slots. Make sure that it slides in all the way to the bottom of the connector. There is a small raised rib on the drawer top which should fit into the recess on the bottom of the pack; if it doesn't, make sure that you have oriented the ROM correctly.

After inserting the ROM, snap the clear cover shut and close the drawer until it is flush with the outside cover of the machine. You are now ready to begin using external mass storage devices.

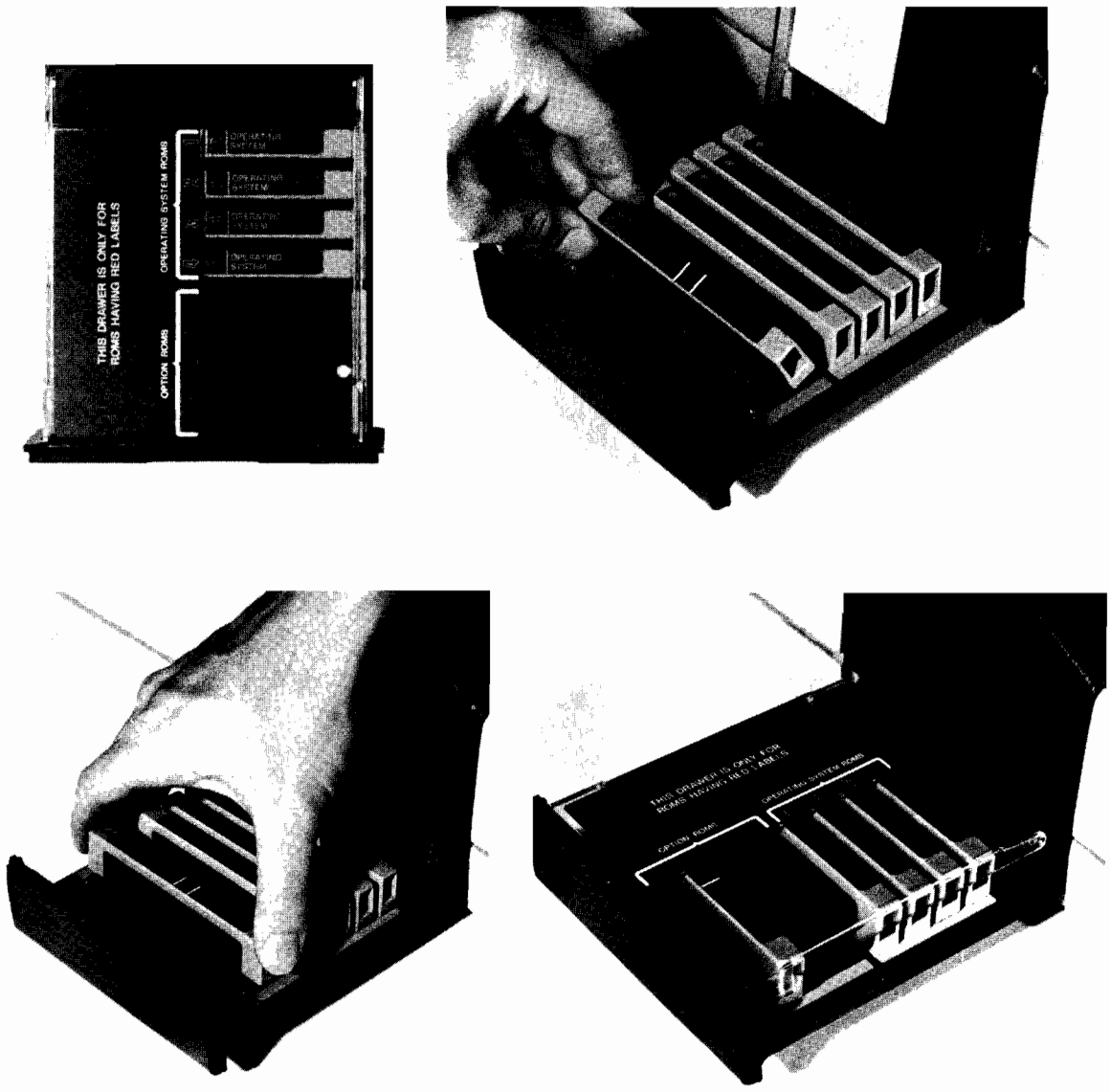


Figure 1. Installing the Mass Storage ROM

Initialization

Before any particular mass storage device can be used with the System 45, the device must be connected and the medium it uses (tape, flexible disk, or hard disk) must be initialized.

The details on how a device is connected, how media are inserted, and other items peculiar to a particular device are treated by the operating manual for the device itself. Please consult the manual(s) that affect you before going on. The proper operation of your mass storage unit is of critical importance to the effective use of the information contained in this manual.

A mass storage medium can be provided by a number of vendors. A list of approved media manufacturers is available through your HP Sales and Service Office (see Appendix F for a list of offices). It is highly advisable that you use *only* HP-approved media on your mass storage devices. Loss of data, damage to the heads, and high maintenance costs may result from the use of non-approved media.

Once a mass storage device is properly connected, a medium inserted, and the system is tested (see the system test booklet – if you are using only the standard tape cartridge, all of this should have been done already), then you must "initialize" the medium itself.

Initialization is a process whereby the System 45 sets up the systems table, directory, and availability table on an individual medium and checks out the records and tracks to assure itself of the recording areas that are available. All of this is so the system can find things when it wants them in the future. It is required that *every* medium used by the System 45 be initialized first. Each tape, flexible disk, and hard disk must go through this process.

With tapes, the initialization process causes each physical record and inter-record gap to be

With a flexible disk, each physical record has a write-check performed upon it and tracks with any defective records are rejected and ignored in future processing.

Error Messages Sticker

Along with the Mass Storage ROM, you received a self-adhesive label (P/N 7120-6835) containing the additional error messages used by the ROM. This label should be placed on the pull-out card underneath the CRT provided for the purpose. This will provide you with a future quick reference for the error messages. A full discussion of the errors and error processing can be found in Chapter 6.

To initialize a medium, first insert it into the mass storage device. Next, execute the statement

```
INITIALIZE msus
```

and wait for the run light to go out, indicating that the initialization process is complete. This should take about three minutes for a tape, between 6 and 21 minutes for a flexible disk, and about two minutes for hard disks. The processing time required is determined by the fact that *every* physical record on *every* track on the medium (tracks only for hard disks) will be accessed and checked.

Interleaving

When initializing a *flexible disk only*, there is an additional parameter called the "interleave factor". It can be added as follows –

```
INITIALIZE msus , interleave factor
```

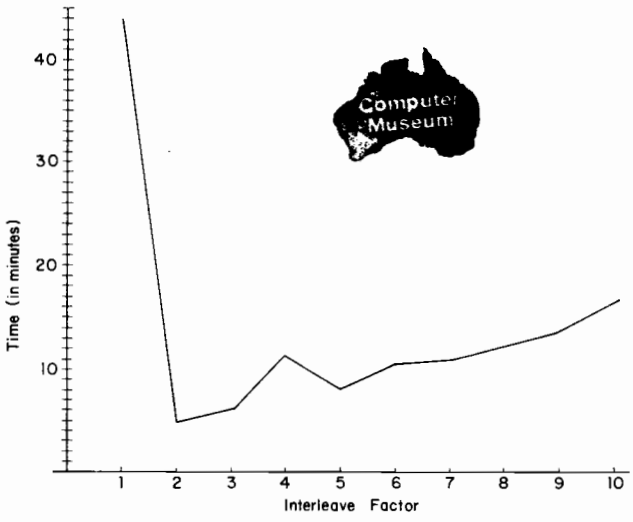
The purpose of this factor is to allow you to control the I/O efficiency of your disk. The **interleave factor** is a numeric expression which must round to an integer in the range of 1 through 10; if you omit it when initializing a flexible disk, it will default to 7. If an interleave factor is used when initializing a tape or hard disk, the factor is ignored.

Interleaving is a process whereby the system effectively renumbers record numbers on a track (so they are no longer consecutively numbered). The records may be numbered consecutively, or by every other one, or by every third one, and so on, up to every tenth one. Because it takes a finite amount of time to read a record, transfer the data to the System 45, and prepare for the next record, and because the disk is spinning for all that time, it is possible for the drive to require a full revolution to read two successive records on the same track.

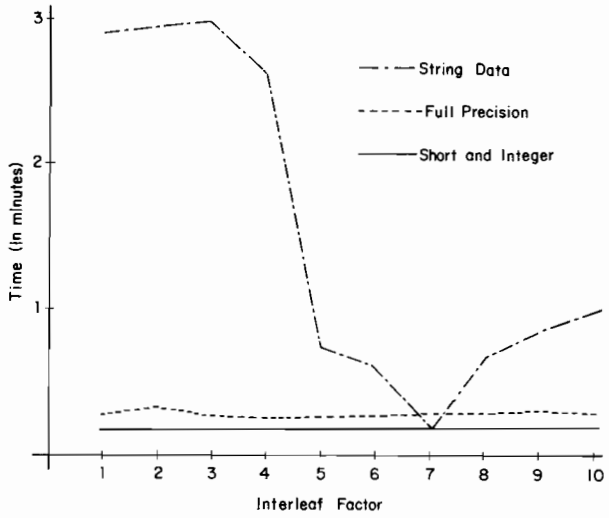
To speed up this process and enable successive records to be read on the same revolution, the interleave factor causes the numbering of records to be altered so that there is physical separation between them, enabling a minimum number of revolutions to be sufficient to read all the records on a track, whereas 30 revolutions might be necessary if they were successively numbered. This can result in access speed improvements of up to 5-to-1.

The following charts detail the disk performance under various interleave factors –

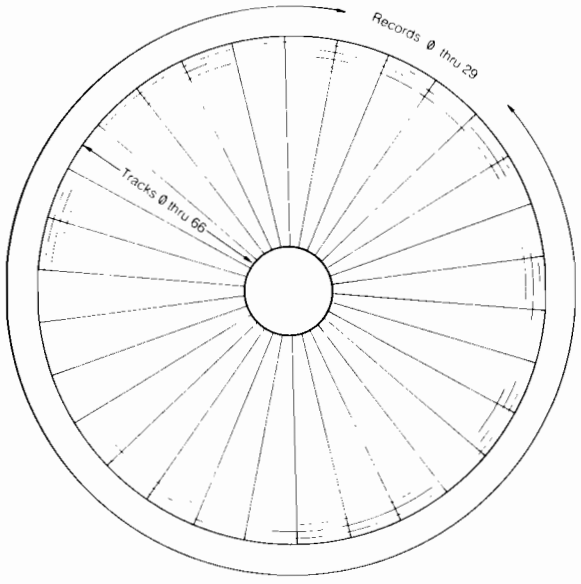
Time Required for Initialization



Writing 1000 Random Records

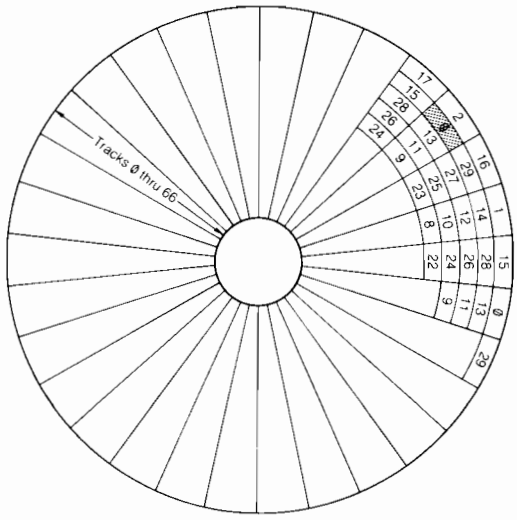


Data is stored in concentric tracks on the disk. Each disk has 67 circular tracks, numbered 0 through 66. The disk is also subdivided into 30 pie-shaped sectors. Each sector contains 67 records (1 record = 256 bytes).



Disk Structure

A diagram of disk tracks and records with an alternating numbering system caused by an interleaving of 2 is shown next. The shaded area shows the location of record 0, track 1, as an example.



Flexible Disk Records

In addition to an alternating numbering system, the location of the beginning record (record 0) of each track is skewed to avoid a revolution when the drive accesses a new track. For example, after record 29, track 0 is accessed, then record 0, track 1 is accessed without an extra revolution.

Flexible Disk Master-Slave System

Whenever using a flexible disk master-slave system, be sure that all units on the same select code are turned on. All must be turned on in order for **any** of the units to be accessed with the statements and commands discussed in this manual.

Data Compatibility with Other Systems

Disks initialized for use on the System 45, and written under the System 45 control, can be used by some other HP desktop computers, particularly the 9831A. However, in defining data files for such transferrance, they must have record lengths of default length (256 bytes). Attempting to use files of different record lengths will cause those machines to become lost on the disk. The System 45-initialized disks cannot be used by the 9825A because of the lack of bootstraps required by that system.

Disks initialized for use on the 9825A and 9831A may be used on the System 45. Data files created by those machines may be used in every fashion the same as if they were created by the System 45. The System 45 will automatically ignore the "bootstraps" area of 9825-initialized disks.

If you are using 9825- or 9831-initialized disks on the System 45, there are some precautions which should be observed –

- Both the 9825 and the 9831 may store values in a disk file which are outside the permissible floating point range of the System 45 (i.e., the absolute value is less than 10^{-99} or is greater than 10^{100}). Such numbers will be read by the System 45 without error and may be used in intermediate calculations and stored in variables. But the numbers themselves, and the results of calculations which are out of range, may not be displayed, printed, or written to mass storage. Thus the lines –

```
100  READ#1;A
110  IF A<1E-99 THEN GOTO 200
120  IF A>9.9999999999E99 THEN GOTO 300
130  ! Data is considered good...
```

would be acceptable and would work correctly, but the subsequent lines –

```
200  DISP A
300  PRINT#2;A
```

would not permit a value out of range to be printed. Instead, if the absolute value is less than 10^{-99} then 10^{-99} would be printed; if the absolute value is greater than 10^{100} then the printed exponent will be unpredictable. Re-packing the file on the 9825 or the 9831 should not be done if the disk contains the System 45-created files of a different defined record length than the physical record length (256 bytes). There should be no difficulties with the System 45 files with 256-byte records. The System 45 files also should not be purged with the 9825 or 9831 if the record length is something other than 256 bytes. (See the following section for an explanation of defined and physical records.)

Storage Operations

Types and Methods of Storage

Mass storage operates through the use of records. Records are the smallest addressable unit of storage. There are fundamentally three types of records – *physical*, *defined*, and *logical*. Records are stored, or grouped, into files. There are six kinds of files – *program*, *data*, *storeall*, *keys*, *binary program*, and *binary data*.

Records

A *physical* record is the unit of storage dealt with by the mass storage devices themselves. They consist on the System 45 of 256 bytes each. As a user, you do not address physical records as such; the bookkeeping and manipulations involving these storage units are handled by the hardware and the operating system.

Defined records are the smallest units of storage which you, as the user, can access individually. The actual length of this record, in bytes, is determined by you in the CREATE statement. This statement is discussed later in the Files section. A defined record may be any size between 4 and 32 767 bytes, but if you select an *odd* number for the record size, it will be rounded up to the next *even* number. All records in a file are necessarily of the same defined size as the one you specify. If you don't specify it, it defaults to a physical record (or 256 bytes).

It is advisable, whenever possible, to set the defined-record size to the physical-record length, in order to achieve maximum efficiency. If there is no program necessity for a different length to be established, selecting 256 bytes over another length can result in I/O performance improvements of as much as 2- or 3-to-1.

Logical records are a structured data concept and have no direct implementation in either the hardware or software of the System 45. These are records in which you *conceive* the data to be organized. In short, it is a collection of individual data items which are conceptually grouped. For example, if you had a personnel application, you might, in the course of programming it, plan a logical record for each employee which contains the employee's number, name, address, department, etc. You might want to deal with this record (like reading it from a data base) as a single record, at least logically. To assist in this application, you might want to define your *defined* records to be large enough to hold all the data in a single logical record. Of course, you could assign the individual components of the logical record to various defined records and keep track of where things are stored. That's up to you. Remember, though, that in dealing with logical records, it is *defined* records which you read and write from storage.

In calculating the length of a part of a record with combinations of numeric variables and strings, it may be helpful to know how much space each of the various types of items take up. To make these calculations, and also to help in determining if the length of a defined record is sufficient, the following numbers apply –

Full-precision numbers	8 bytes each
Short-precision numbers	4 bytes each
Integers	4 bytes each

Strings are a little more complicated to calculate –

1 byte for each character
1 more byte if the string length is odd
4 more bytes ("overhead") for each defined record in which it resides (it must be in at least one – some stored strings cut across defined record boundaries)

When you are dealing with arrays, you are dealing with individual elements, the number of which is equal to the current working dimensions (according to the current OPTION BASE selected). Each element is of the length indicated by the array's type (full, short, integer, or string).

No extra bytes are stored to identify information as having been written from an array; rather, the arrays are written so that it appears in storage as a sequence of individual elements. Thus it is possible to write something as an array and later read back the information as individual elements into non-array variables. Conversely, it is also possible to read a compatible sequence of individual items into an array, even if they were originally written as non-array items.

To give some examples of this accounting, suppose the following items were being stored or retrieved; also suppose F is a full-precision variable, S is a short-precision variable, I is an integer variable, and S$ is a string variable (with a current length of ten characters) –

F	requires 8 bytes
I,S	requires 8 bytes
S$	requires 14 bytes[1]
S$,I	requires 18 bytes[1]
F,S$,S	requires 26 bytes[1]

[1] provided the string doesn't cross a defined-record boundary.

Files

Mass storage is organized around the concept of a *file*. A file is a common collection of records. As such, it is a contiguous grouping of storage locations on the storage medium. There is a "directory" at the beginning of each medium which gives the name, length, and type of each file on it.[1] Each file is in one particular location on the medium and all of its records are in order. The directory also contains the location of the first record in each file. In addition to the directory, there is an "availability" table and systems table. Each medium, then, is organized something like this –

SYSTEMS TABLE
DIRECTORY
AVAILABILITY TABLE
FILE #1
  1st record
  2nd record
  3rd record
    .
    .
    .
  Last Record
FILE #2
  1st record
  2nd record
  3rd record
    .
    .
    .
  Last Record

FILE #3
  .
  .
  .
LAST FILE

Since files may be created and "purged" with some frequency, gaps may develop in this scheme so that there are a number of unused physical records between files. The problem of "wasted" space of this type is dealt with in Chapter 3.

Where each record is physically located is highly dependent upon the device itself. However, accessing the file itself is totally independent of the type of device. To get a particular file on a device takes only its file name and an **msus** as discussed above. To find the file, the System 45 goes to the device indicated by the **msus** part of the **file specifier** and looks at the directory of the medium on that device for a stored file with the same file name. If such a file exists, then it will take note of the location (track and physical record) of the first record of the file with that name. All other records in the file will be displaced from the first record. For example, the eighth record in such a file will be seven records past the first record, and so on.

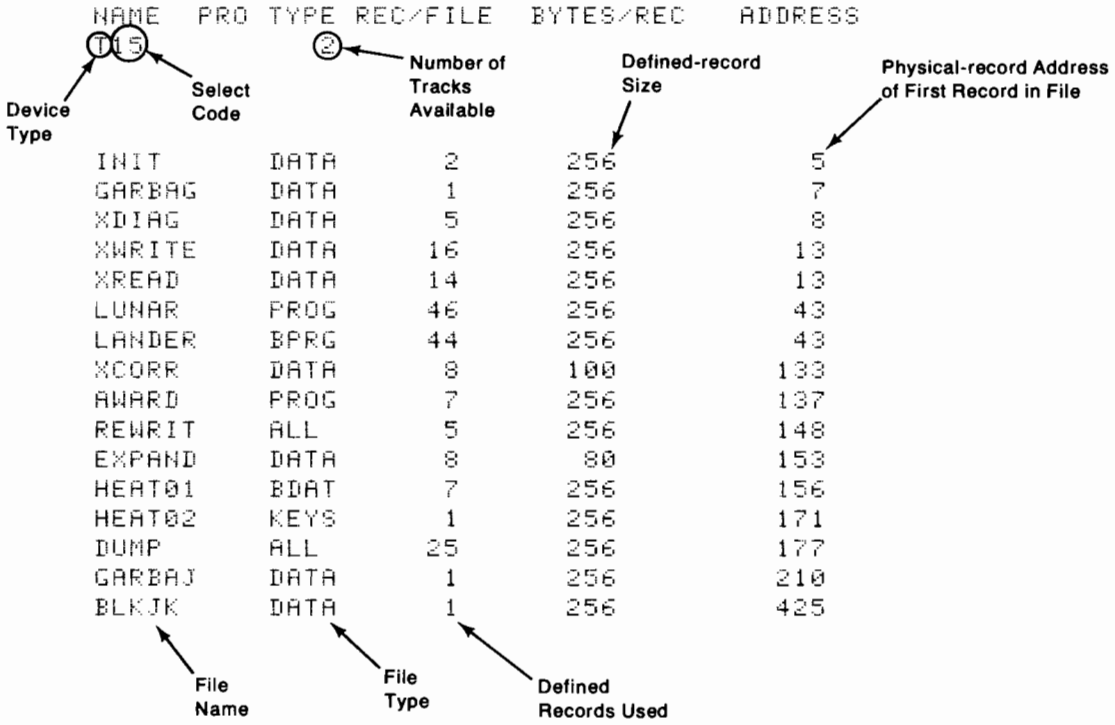
1 A flexible disk with an interleave different from 1 spreads out the actual physical location, but the system still takes records and files in (interleave) order.

File Directory

A directory, as mentioned above, is present on each medium (not the device). Replacing the medium in a device will change the directory (and, of course, the files) available to you. The information in the directory may be listed on the system printer. The statement to use to do this is –

CAT [selective catalog specifier / msus] [, heading suppression]

The following is an example of a directory listing using the CAT statement on a tape –



The **heading suppression** parameter is a numeric expression. If it evaluates to 1, the heading is suppressed. If it contains any other value, then the headings will remain. This enables you to print selective catalogs one after another, but to have only one heading.

The **selective catalog specifier** is a string expression which gives you the capability of selectively listing parts of the catalog. If the parameter is present, the catalog routine will only list those files whose names begin with the same string value as the parameter. For example, if the above directory had been listed with the statement –

CAT "X"

the following would have been the listing –

```
NAME   PRO TYPE REC/FILE   BYTES/REC    ADDRESS
T15           2

XDIAG      DATA       5       256           8
XWRITE     DATA      16       256          13
XREAD      DATA      14       256          29
XCORR      DATA       8       100         133
```

and the statement

```
CAT "GAR",1
```

would have produced –

```
NAME   PRO TYPE REC/FILE   BYTES/REC    ADDRESS
T15           2

GARBAG     DATA       1       256           7
GARBAJ     DATA       1       256         210
```

The statements –

```
CAT "A"
CAT "B",1
CAT "D",1
CAT "E",1
```

would have produced –

```
NAME   PRO TYPE REC/FILE   BYTES/REC    ADDRESS
T15           2

AWARD      PROG       7       256         137
BLKJK      DATA      69       256         211
DUMP       ALL       25       256         177
EXPAND     DATA       8        80         153
```

A catalog will normally be printed on the standard system printer, whatever that happens to be at the time the CAT statement is executed. But it is possible to divert the listing to another printing device which is on-line at the time. This is done by adding the device's select code (and its HP-IB address, if applicable), so that the syntax of the statement now appears as –

CAT # select code [, HP-IB device address] [; selective catalog specifier / msus [, heading suppression]]

where **select code** and **HP-IB device address** are numeric expressions.

For example, if the statements in the above example had been –

```
CAT#16;"A"
CAT#16;"B",1
CAT#16;"C",1
CAT#16;"D"
```

the same listing as the above would have been produced, but on the CRT (which is select code 16).

Another example, using a flexible disk –

```
NAME   PRO TYPE REC/FILE  BYTES/REC   ADDRESS
F8           65
BLKJK       DATA      69        256       34/14
SEN         DATA      35         58       33/14
EXPAND      DATA       5        256       31/15
DATA        DATA       4        256       21/01
COPY        DATA     100        256       21/05
REP         DATA      65         58       33/22
GEORGE      DATA       6        256       33/08
XWRITE   *  DATA       7        256       34/07
```

Record Address

File Protection

Track Address

In this example, the address indicates the track position, as well as the record position, of the first physical record of the file. Also, it demonstrates that an asterisk will appear in the column after the file name for any file which is protected.

Types of Files

You may have noticed in the file directory that there may be more than one type of file. In actuality, there are six types of files –

Type	Directory Abbreviation
Program	PROG
Data	DATA
Storeall	ALL
Keys	KEYS
Binary Program	BPRG
Binary Data	BDAT

Each is created in a different manner, and each serves a different function. Since each of these can be used in some applications and not in others, selection of a file type will depend upon the application. In general, these files have the following properties –

PROGRAM–Created by a STORE or RE-STORE instruction; retrieved by a LOAD instruction. Stores a program in its internal representation (compiled form), along with its cross-reference tables.

DATA–Created by a SAVE, RE-SAVE, or CREATE instruction; retrieved as a file by a GET or LINK instruction, or by individual records with READ# statements. Used to store programs in "source" form or data written by PRINT# statements.

STOREALL – Created by a STORE ALL command; retrieved by a LOAD ALL command. Stores the entire machine state – memory, keys, etc.

KEYS–Created by a STORE KEY instruction; retrieved by a LOAD KEY instruction. Used to keep the definitions of the special function keys so that they may be reloaded (and hence restored to some previous form).

BINARY PROGRAM–Created by a STORE BIN instruction; retrieved by a LOAD BIN instruction. Used to store special HP-supplied routines which implement special enhancements to the standard BASIC language.

BINARY DATA – Created by an FCREATE instruction; individual records are created by FPRINT instructions and retrieved by FREAD instruction. Used by DMA mass storage devices (but not the internal tape cartridge) to do fast transfer at DMA speeds.

Types of Access

There are two methods (or "modes") of data access with the System 45 – serial and random. DATA files – and DATA files only – may be written and read using either access method. Most applications reference a particular file using one mode or the other exclusively, but it is entirely possible to write a file in one mode and read it back in another.

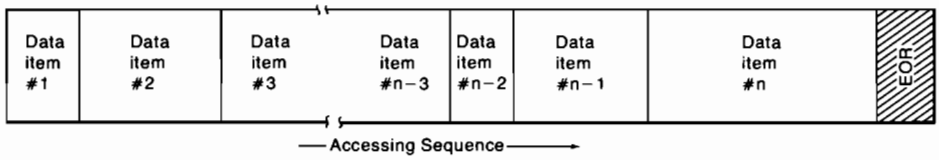
Each mode of access has advantages and disadvantages over the other depending upon the application and mass storage device being used.

Serial Access

Serial access is a mode which relies upon the *sequential* nature of the data in the file. Data items are read or written one after another. With serial access, logical records may be of varying lengths. The last data item written by a serial PRINT# statement (discussed below), is followed by a one-byte end-of-record (EOR) mark. After the EOR is written, the file pointer is positioned *at this mark*.

When writing in the serial mode, you begin writing immediately at the file pointer. This has the effect of writing over the EOR mark which may be there. Each item, then, immediately follows the one before it and has *no* mark separating the two. Thus, a serially-written file is merely a sequence of data items, with no way to tell where one record ends and the one after it begins. This is where the concept of a logical record helps to keep the location of things straight.

Serial access, therefore, is a compact method of data storage. There is a minimum of wasted space. The succession of data items, one after another, is called the "accessing sequence". The storage structure for "n" data items in serial-access looks like this –

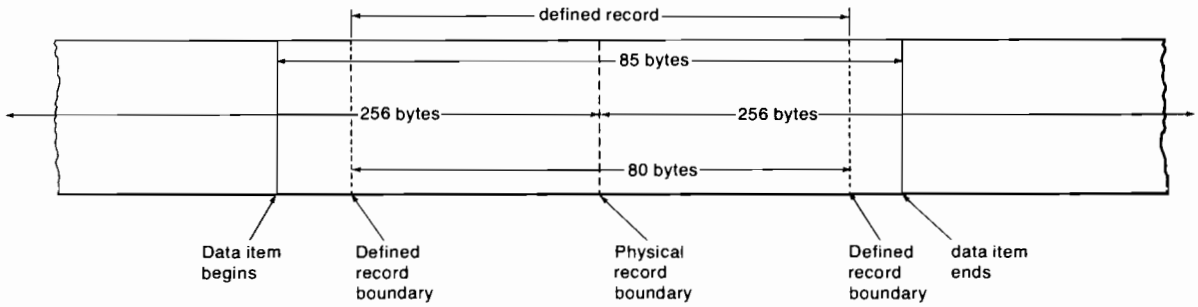


As data is written in a serial file, previous end-of-file (EOF) marks which may have existed there are written over as they are encountered. An exception to this is made when the EOF mark at the beginning of a defined record would be written over by an EOR. Instead of doing this, the EOF mark is allowed to remain. The file pointer will still be positioned there, however, and the next serial write will treat the EOF as an EOR and write over it.

The variable-length nature of the records means that serial access ignores defined-record boundaries as well as physical-record boundaries.

If you are going to be using a serially-written file strictly as a serial file, then there is no need for you, as a programmer, to have to be concerned about such boundaries. No boundaries (except the physical end of the file), have any program effects.

For example, if you were to write a serial access record of 85 bytes to a file which has defined records of 80 bytes each, then obviously it would take more than one defined record to hold the information. Therefore, you would pass over at least one defined-record boundary. The following diagram demonstrates what this might mean –



You will not notice the physical, or even the defined, record boundaries. If you were later to read back those 85 bytes in serial mode, you would *not* have to account for the presence of the boundaries – only the original 85 bytes would be returned to you.

Serial access is sequential in nature. When you write a logical record, as was noted earlier, it is written to a location which immediately follows the previous logical record written to the file (provided there remains sufficient room in the file to hold it). Thus, records in a serial access file appear in a certain order, namely the accessing sequence, i.e., the order in which they were written.

Sometimes an EOR mark written after a data item will be permitted to remain in the file. This occurs whenever the next item to be written in a serial mode is too long to fit in the remainder of the defined-record (strings are excepted). For example, if six bytes remain in a defined-record after the last write operation (including the EOR mark), and you try to write a full-precision number (which takes eight bytes), the EOR mark will be left there and the number will be written at the beginning of the next defined-record. This is so the system won't try to divide indivisible numeric data (i.e., full-precision, short-precision, and integer) between records.

For strings, which can easily be longer than a defined-record, the rule above is modified somewhat. If four bytes or less remain in a record and you try to write a string, the EOR will remain and the string will begin with the next record.[1] If more than four bytes remain, it will put as many characters as it can in the record (there is an overhead of four bytes), and continue the string in the next record, if necessary.

[1] A null-string will fit in four bytes, since there are no actual characters to write. The four-byte overhead will be written, though, to indicate the presence of the null-string.

You must be careful in the selection of defined-record sizes. If you make a poor selection of record size, you can significantly waste storage space. For example, if you chose a defined-record size of 14, but then wrote serially a number of full-precision numbers (8 bytes each), and nothing in between, you would be wasting 6 bytes per record. This would be effectively wasting 43% of the storage space in the file. Thus, it is a highly advisable that shorter defined-record lengths be selected very carefully to match the data to be written, or the wasted space can be devastating.

In a file used strictly in serial access, there is no reason to have defined records of any length other than physical-record size (256 bytes). There are very significant speed improvements when records are of that length.

Data in a serial-access file must be read in the accessing sequence. If you are interested in reading the data in the n^{th} data item written, you must first read the preceding $n-1$ data items in that file. Serial reading will ignore EOR marks left in the file, so you do not, as a programmer, have to take them into account in your serial reading.

Serial access is the ideal method for use with tape cartridges since they are essentially sequential media.

Serial Access Summary

- Logical records may be of varying length.
- Data is a sequence of items with no marks separating them; i.e., data are read and written sequentially.
- The last data item written is followed by an EOR mark; this mark is destroyed by the next data item written at this point unless there is insufficient room in the defined record for that item.
- Defined-record boundaries and physical-record boundaries, as well as EORs left in the file, may be ignored for most programming purposes.
- Defined-record size affects storage efficiency and program performance.
- Ideal method for tape cartridges.

Random Access

Random access is a method which relies upon the fixed-length nature of all records in a file. Defined records may be written and read in any order. With random access, data read or written must be of a certain maximum length. Each defined record is assigned a "record number" for access purposes.

Random access files utilize the defined-record feature of the file's creation to advantage. Records in random access are precisely equivalent to the defined records of the file. A normal data file is a random access file, just as it is also a serial access file. Thus, by creating a file of 100 80-byte defined-records, you are creating a file of 100 80-byte random access records as well.

These records are accessed by a "record number". This may be any numeric expression which, when evaluated and rounded to an integer, arrives at a positive number less than or equal to the number of records in the file (you can't access a record which doesn't exist).

For example, if A = 10 and B = 4, and we had a file of 100 defined records, the following would be proper as a **record number** –

```
33
A+1
100-B
A*9+B+1
```

but the following would be *improper* –

```
101       (greater than the number of defined records in the file)
A-(3*B)   (a negative number)
A/10-1    (zero)
```

Each random access record is exactly the length of a defined record in the file. Hence, it is not possible to write anything in random access unless it has the same number of bytes (or less) than a single defined record. It is this which allows the random access method to work as it does.

The method implies that each record is accessed in the same manner as every other record and that you need not access all preceding records in order to get to the one you desire, as you must do when the access method is serial.

There is no "accessing sequence" as there is with serial access. You may access records in any order. But in order to achieve this property, the random mode requires each record to be of a *fixed* length, whether you use all of it or not. You may end up (possibly) wasting some space within a file if you do not use all the available space in each record. Of course, it is possible to select a record size upon creation of the file with an eye toward minimizing (or perhaps even eliminating) potential waste of this type.

Random access is a good method for use with disks (both flexible and hard), which are not sequential media, but it is not a particularly good method for longer files on tape.

For disks, each record takes approximately the same amount of time to access, thus widely separated records take no longer to access than do records which are closer together. With tapes, however, if records are physically far apart, it may take significant time to access them both, but if closer together, the records can be accessed relatively rapidly. For this reason, it is recommended that if you are accessing random records with widely separated numbers, that you avoid the use of tapes.

Random Access

Summary

- Records within a file will be of the same length (though you may use only part of any given record).
- A record number is used to specify the record being accessed.
- Reading and writing operations may be intermixed.
- Each record corresponds to a defined record.
- Good method with disks; not so good with long files on tape cartridges.

Creating Files

As was pointed out earlier (page 24), there are a number of statements which create files. In all except the DATA and BINARY DATA types of file, the contents of the file are created when the file is created. Only with the DATA or BINARY DATA file types can the file be created separately and remain "empty" until needed for storage purposes.

BINARY DATA files, and their creation, are treated in "Rapid Transfer of Arrays" below.

Storage space on a medium can be reserved for a DATA file through the CREATE statement. The statement can be used only to create DATA files and can be programmed or entered from the keyboard. It is used to establish the length and number of the *defined* records in the file. It is only through this statement that you have control over the defined records. With all other statements which require creation of a file, the defined records become equal to physical records.

The CREATE statement appears as follows –

CREATE file specifier , number of defined records [, record length]

number of defined records and **record length** are both numeric expressions, which, when evaluated and rounded to an integer, become the number and length, respectively, of the defined records in the file. Consequently, in order not to be nonsense, these values must be positive numbers. They may not extend beyond the capacity of the medium, however. The maximum number of records which may be specified in any file is 32 767. The maximum length is 32 768 bytes. The minimum length of a record is four bytes. The maximum number of physical records which may be used by a file is 32 767.

If you give an odd number of bytes for **record length**, it will be rounded up to an even number. If **record length** is omitted, the defined-record length for the file will default to 256 (same as the physical-record length).

Some examples –

```
CREATE "Volts",100,10
CREATE "Amps",24
CREATE "Trans:F",100,80
CREATE  File$,1000,Length
```

Creating a file causes two things to happen. First, it is entered in the directory. All associated information – defined-record size, number of records, physical-track and -record location of the first record, file type, and protect code – are all stored in the directory. Every file starts at the beginning of a physical record, so there may be some wasted space between files (though not more than 254 bytes each time).

Second, the system causes an EOF mark to be placed at the beginning of every defined record in the file. Later on, when you are using the file, you will write over some of these marks (if not all). By having initially written one of them to every record, you are guaranteed of having an end-of-file indicator somewhere in the file, regardless of whether you write to it in serial or random fashion. In addition, this procedure effectively clears each defined record so that no old data remains from any previous use of the record.

Record I/O

During several points in the discussion in this chapter, there has been mention of "writing" and "reading" records to files. This is also called "record I/O". Record I/O is available only on DATA-type files. All other types of files must be manipulated as a whole.

Before record I/O can take place on a file, you must assign a "file number" to it. There are 10 file numbers available for your use, 1 through 10. To assign a file number to a file, you may say either –

ASSIGN # file number TO file specifier [, return variable [, protect code]]

or –

ASSIGN file specifier TO # file number [, return variable [, protect code]]

file number may be any numeric expression which will round to an integer between 1 and 10.

return variable may be any numeric variable. After execution of the ASSIGN, the return variable will contain a value indicative of the status of the file, thus indicating if an error occcurred during the ASSIGN. The values which may be returned are –

Value	Meaning
0	File exists and is available for your use
1	File could not be found
2	File was protected, or was of the wrong type or **protect code** included when file does not require it.

protect code is a string expression, which is necessary only if the file had previously been protected by a PROTECT statement. See Chapter 3 for details. If the **return variable** and **protect code** are omitted and the file is protected, there will be an error. There will also be an error if you omit the **return variable** and the **protect code** is present, but the file is not protected.

This assignment must be done to any data file before you can read from it or write to it. Of course, the file must have already been created. A file number may be used any number of times in a program and may be re-assigned to another file at any time by simply executing another ASSIGN. For example, in this program –

```
100  ASSIGN #1 TO "Keep1"
 .
 .
 .
500  ASSIGN #1 TO "Keep2"
 .
 .
 .
1000 ASSIGN #1 TO "Keep3"
 .
 .
```

file #1 was first assigned to Keep1, then to Keep2, and finally to Keep3. Each reassignment cancelled out the one before it. This happens upon *execution* of the ASSIGN statement. Thus, if after line 1000, the program looped back to 100 (say by a GOTO 100), #1 would be reassigned to Keep1 again.

This file number is used in the PRINT# and READ# statements which are the ones used to effect record I/O in mass storage. Once a file number has been assigned in this fashion to a file, then referring to the file number is the same as referring to the file, though it does not take the place of a file specifier in a syntax.

Executing the ASSIGN statement also flushes any file buffer allocated to the file number (by the BUFFER statement). For further details on this effect, and on buffers in general, see Chapter 4. The statement also sets the data pointer to the first byte of the first defined record in the file.

More than one file number may be assigned to a single file. This permits more than one file pointer in the same file and references to the separate file numbers is the same as references to different files. Thus, with this facility, you could read and print in different modes without disturbing the file pointer in each with operations using the other. This facility can also improve buffer performances.

Using the ASSIGN statement as above causes a data file to be "opened", that is, to make it available for use by PRINT# and READ# statements. When you want to "close" a file, that is to make it unavailable for use by I/O statements, you also use the ASSIGN statement.

There are two ways to use the ASSIGN statement to close a file. The first is to re-assign the file number to another file. Thus, in the example above, execution of line 500 not only re-assigned the file number (#1) to Keep2, but it closed the file to which #1 had been previously assigned.

The second way is to use the asterisk character (*). Appearance of this character in lieu of the file specifier will cause the file assigned to the file number to be closed. For example –

```
100  ASSIGN #1 TO "George";Return
 .
 .
 .
500  ASSIGN #1 TO *
 .
 .
 .
```

execution of line 100 will cause file #1 to be associated with George. Thus George is opened. But subsequent execution of line 500 will cause #1 to become unassigned and George to be closed. Further attempts after execution of line 500 to do a PRINT#1 or READ#1 causes an error (error number 51) unless there is another ASSIGN #1 executed.

If a file is opened in a subprogram (either a SUB or a multi-line function), then upon returning from the subprogram, the file is closed, the same as if ASSIGN * had been executed for that file number.

ASSIGN is always performed in the SERIAL (non-overlapped) processing mode, regardless of whether overlapped processing is currently in effect. Overlapped processing for other statements is not affected by the presence of an ASSIGN statement.

CAUTION

WHEN USING A DISK SYSTEM WITH MORE THAN ONE UNIT ON A SELECT CODE, DO NOT RE-ASSIGN A FILE NUMBER FROM ONE UNIT TO ANOTHER USING THE SAME SELECT CODE UNLESS YOU FIRST CLOSE THE FILE. RE-ASSIGNING A FILE WITHOUT CLOSING IT FIRST IN SUCH CIRCUMSTANCES WILL CAUSE A SYSTEM DEADLOCK.

Writing Records

To *write* a record in the *serial* mode, the statement to use is –

PRINT # file number [; data list]

Upon execution of such a statement, the **data list** (which consists of variables, constants, arrays, etc., the same as in PRINT statements to a printing device, except that all items are separated by commas only – not semicolons – and none of the output functions TAB, SPA, LIN, and PAGE may appear) is moved to the file indicated by the file number and written, starting at the place indicated by the data pointer. There are only three errors you can make with this statement –

- Using a **file number** not assigned to an existing file.
- Running out of room on the file.
- Hardware not working, or medium not present.

To *write* a record in the *random* mode, the statement to use is –

PRINT # file number , record number [; data list]

Notice that this form of the statement is quite similar to the serial form. The only difference is the addition of the **record number**. In this case, the data confined in the **data list** will be moved to the file specified by the **file number** and to the particular record specified by the **record number**. There are a number of ways one can err using this form. The rules involving random access prints are –

- The **file number** must be assigned to an existing file.
- The record indicated by **record number** must exist on the file.
- The data in the **data list** must not exceed the size of the defined record (it may be less, however).

Omitting the **data list** will cause an EOR to be written at the current position of the file pointer. This causes any data in the remainder of the defined record to become inaccessible.

Print Verification

In many applications it is critical that the data written to a mass storage device be as accurate as possible. The CHECK READ statement is a way to increase significantly the reliability of data written to mass storage.

Using the CHECK READ statement for a particular file instructs the System 45 to check every item it writes to that file. It does this by immediately following each PRINT# operation to the file with a read in the same area of the medium which it has just written (called a "read-after-write" operation, or "verification"). The results of this read are compared with the contents of memory for the data written. If they compare identically, the PRINT# is considered successful. If they are not identical (implying there has been some sort of failure, either on the write or the read-after-write operation), it will try writing and verifying the physical record three more times before giving you an error message indicating that you have encountered a problem.

The error generated by a CHECK READ is a fatal error (number 89) and will cause your program to stop unless trapped by an ON ERROR statement. The ERRL, ERRN, and ERRM$ functions will also work with this error. (More information on these functions and the ON ERROR statement can be found in the operating and programming manual.) The ON ERROR trap will not work on CHECK READ if you are processing in the OVERLAP mode (see Overlapped Processing, pages 66-67).

Check reading significantly slows the PRINT# operation. For tapes, the verification process causes three passes to be made over the tape (the original PRINT# operation, backing up the tape over the record, then the reading operation). For flexible disks there must be a full

revolution of the disk after each PRINT# in order to make the read – in contrast to writing many records per revolution. The consequent decrease in access speeds can be as much as 5-to-1.

In addition to a decrease in operating speed, there can be an increase in wear of the medium itself with tapes and flexible disks, in those spots where the heads come into actual contact with the medium.

NOTE

With tapes and flexible disks, because of the combined problems of decreased speeds of operation and increased wear on the media, it is recommended that CHECK READ be used only when data correctness is of paramount concern. On hard disks, these problems are not significant, and the CHECK READ can be used without worrisome side-effects.

The CHECK READ statement has the following form –

CHECK READ [#file number]

where **file number** is a numeric expression representing the assigned number of the file involved. If no **file number** is provided, CHECK READ will be in effect for all printing operations to *every* mass storage device.

Some examples of enabling CHECK READ –

```
80   CHECK READ #1
90   CHECK READ #2
100  CHECK READ #3

500  FOR I=1 TO 5
510  File$="Data"&VAL$(I)     ! Naming convention
520  CREATE File$,100,80      ! Create the file
530  ASSIGN File$ TO #I       ! Assign the file a number
540  CHECK READ #I            ! Enable CHECK READ on the file
```

An enabled CHECK READ for any file number can be turned off by closing the file (see page 32),or by using the CHECK READ OFF statement for that file number, which has the form –

CHECK READ OFF # file number

Again, if the **file number** is omitted –

```
CHECK READ OFF
```

then executing the statement has the effect of turning off CHECK READ for all files.

In addition to the verification of data written with a PRINT# statement, by omitting a file number from the CHECK READ statement, you also permit all file operations (except FPRINT) to be verified. After executing a CHECK READ, any SAVE, RE-SAVE, STORE, RE-STORE, STORE BIN, or STORE ALL statement will also be verified. Executing a CHECK READ OFF statement cancels this facility.

CHECK READ also has the effect of causing an immediate-write for a file on a device, flushing the device buffer. For further details on buffers and on this effect, consult Chapter 3.

Executing CHECK READ without a file number will necessarily enable CHECK READ for *all* files, and they cannot be turned off separately in this instance.

Upon power-on, and SCRATCH A, all CHECK READs are turned off.

Reading Records

Reading a record from mass storage is a little more complex than writing one, but not by much.

To *read* a datum in a *serial* mode, you must have read all data which precedes it in the accessing sequence. Upon assigning a file number to a file, you reposition the accessing sequence pointer (also called the file pointer) to the first datum in the *file*. Data may then be read by a statement in the following form –

```
READ # file number ; variable list
```

The **variable list** is of the same type as variable lists used in the non-mass storage READ and INPUT statements (see the Operating and Programming Manual).

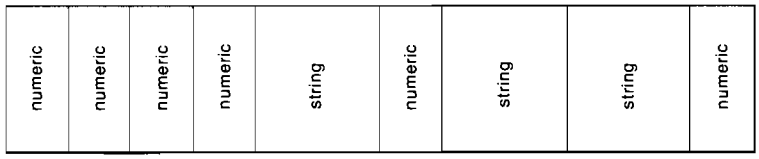
The special rules relating to mass storage reads are –

- The **file number** must be assigned to an existing file.
- Variables in the **variable list** must agree as to data type with the information it tries to read in the record (a string may not be read into a numeric variable, and vice-versa). Disagreements between numeric data types (full-precision, short-precision, and integer) are automatically converted, but there is a possibility of precision losses or overflowing the range in such cases.
- There must not be more variables in the list than there are data items in the record.

In reading a file serially, you are reading data items, and not records. You take the data as it comes, and may do so in any order consistent with the data types. An example will alleviate the confusion regarding this process. Suppose a serial file was created by executing the following PRINT# statements in this order –

```
PRINT #1;A
PRINT #1;B,C,D
PRINT #1;A$,E
PRINT #1;B$,C$
PRINT #1;F
```

Then the file would have the following structure –



Later, if you were to "rewind" the file (reposition the pointer back to the beginning, either by re-assigning the file number or by running another program, or by executing a random READ# to record #1; see page 40), and tried to read the data in this file, the following would be a perfectly valid sequence of READ# statements for that purpose –

```
READ #1;M,N,P
READ #1;Q,R$,S
READ #1;T$
READ #1;U$,V
```

still another valid way might be –

```
READ #1;M,N,P,Q,R$,S,T$,U$,V
```

To *read* a record in a *random* mode, you must provide a record number to indicate which record you are reading. The form of the READ# statement in this case is –

```
READ # file number , record number ; variable list
```

The rules for this form are the same as with serial reading, with the addition of –

- The record indicated by **record number** must exist on the file.

In this form, you read data starting at the first item *in the record*. And you may not read more data items than there are present in that defined record. Thus, if a defined record in a random file were written with the following –

```
PRINT #1, 10; A, B$, C, D$
```

then it is only possible to read a maximum of four data items from record #10: a full-precision number, a string, then another full-precision number, and then another string. Thereby –

```
READ #1, 10; X, Y$
```

would be perfectly valid for this record, but –

```
READ #1, 10; W, X$, Y, Z$, A
```

would not work at all.

By way of example of the CREATE, ASSIGN, PRINT#, and READ# statements just presented, the following programs demonstrate how you might use these statements to copy certain files from tape to disk:

A serial access file containing string variables only, each up to 252 characters long –

```
10   DIM A$[252]
20   CREATE "FARADS:F",100
30   ASSIGN #1 TO "FARADS:T"
40   ASSIGN #2 TO "FARADS:F"
50   FOR I=1 TO 100
60   READ #1;A$
70   PRINT #2;A$
80   NEXT I
90   END
```

A random access file where each record is a single string and the original file is 100 80-byte records –

```
10   DIM A$[76]
20   CREATE "OHMS:F",100,80
30   ASSIGN #1 TO "OHMS:T"
40   ASSIGN #2 TO "OHMS:F"
50   FOR I=1 TO 100
60   READ#1,I;A$
70   PRINT #2,I;A$
80   NEXT I
90   END
```

You can also position the file pointer to the beginning of a defined record without actually doing any I/O operation. This is done with a READ# statement with the data list omitted. For example –

```
140  READ#1,10
```

has the effect of placing the file pointer for file #1 at the beginning of defined record 10. Later accesses to the file, would then begin at this point. An application using this capability can be found in the next section.

Using Serial and Random Access Together

It is entirely possible for you to access a file in one mode and then switch to another mode for later accesses. For example, here is an application where random access is applied to position the file pointer in the file, then a logical record is read and printed with serial access –

```
100  ASSIGN #1 TO "PAYROL"
110  INPUT "Employee number?",Number
120  READ #1,Number
130  READ #1;Name$,Rate
140  INPUT "Hours?",Hours
150  PRINT #1;Hours
 .
 .
 .
300  GOTO 110
310  END
```

Rapid Transfer of Arrays

In addition to the above methods of data transfers, all of which are equally valid and appropriate for arrays, there is an additional method for the transfer of data stored in arrays to *non-tape* mass storage devices. It permits data transfer to occur at direct memory access (DMA) rates – i.e., at the maximum speeds permissible by the operating specifications of the hardware. To help accomplish these speeds, the device buffer is ignored in the transfer and data is transmitted directly to or from the device.

Such transfers are only possible with arrays transferring to and from a special data file called the Binary Data file (called BDAT in the directory listings). This file must be created before it can be used. The statement needed to create it is –

FCREATE file specifier , number of physical records

where **number of physical records** is a numeric expression for the number of physical records desired in the file. Note that there can only be physical records, and no defined-record size parameter is present. It is a feature which permits the statement to work as fast as it does.

To calculate the number of physical records needed in the file, the following formulae are used –

- For string arrays –

  INT [number of elements × (bytes per element +2)/256] +2

  where the **bytes per element** is the *dimensioned* length of each element, and *not* the current string length of the individual elements. Add one additional byte if the dimensioned length is odd.

- For numeric arrays –

  INT [number of elements × k/256] + 2

  where k = 2, if an integer array; k = 4, if a short-precision array; or k = 8, if a full-precision array.

Don't forget in such calculations to take into account the OPTION BASE you are currently using when counting the number of elements.

The first record of the file is an "overhead" used to store information needed by the system.

The maximum *useful* size of BDAT files is 257 physical records. The minimum size is 2.

Individual arrays may be written to, or read from, a Binary Data file. Only entire arrays can be written or read. No simple variables, or constants, are allowed in the reads and prints to such a file, and only one array may be read or written to a file.

To write an array to such a file, the statement to use is –

FPRINT file specifier , array identifier

and to read such an array the statement is –

FREAD file specifier , array identifier

file specifier is the file specifier for the BDAT file being used. You do not "assign" BDAT files to a file number as you do usual data files. The **array identifier** assures that you are writing or reading only an entire array. Individual elements are not allowed. The array being used must be of the same data type (integer, full-precision, short-precision, or string) as the original array written to the file.

When storing an array with FPRINT, the current dimensions of the array are stored along with the data. When reading the data back with FREAD, the array being used to receive the data must have the same *number* of dimensions (e.g., a two-dimensional, a three-dimensional, etc.), but not necessarily the same number of elements in each dimension, as the original array used in FPRINT. Upon reading the data, the receiving array will be re-dimensioned to the current dimensions of the original array. It is important, then, that the receiving array be large enough in number of elements, as well as the number of dimensions to allow the re-dimensioning to take place. See the example below for a demonstration of this effect.

String arrays will also be re-dimensioned with FREAD as above, but the string length of each element must be identical to the original used in the FPRINT. Thus, if you FPRINT a string array where the elements are dimensioned to 80 characters, then you must FREAD the array into a string array which has elements dimensioned to 80 characters.

Since both FPRINT and FREAD make a reference to the file each time they are executed, the file pointer with such statements always begins with the first record in the file. Consequently, you are only able to read and write a single array to a BDAT file. Subsequent FPRINTs, for example, to the same file will simply write over the data which was there previously.

Speed improvements (as much as 25:1) over the ordinary PRINT# and READ# statements are dependent upon the type of mass storage device being used. This technique cannot be used with the internal tape cartridges (T14 and T15).

Since the object of an FPRINT is rapid transfer, data is *not* verified via the CHECK READ which may be in general effect (see page 36). The execution of an FPRINT should not affect the verification for subsequent executions of PRINT# statements.

Here is an example of the use of all three statements –

```
10   DIM A(5,20),B(10,10),C(10,5),D(10,5)
20   File$="RANDOM:F"
30   FCREATE File$,100*8/256+2  ! Create BDAT file big
40   RANDOMIZE                  ! enough to hold A(*)
50   FOR I=1 TO 100
60   A(I)=RND
70   NEXT I
80   FPRINT File$,A(*)          ! Store the random numbers
 .                              ! in BDAT file
 .
 .
200  FREAD File$,B(*)           ! Retrieve the random
210  MAT D=B*C                  ! numbers in a new array
```

If you are printing or reading arrays to a protected BDAT file, then you must include a protect code (see "Protecting a File" in Chapter 3 below).

FPRINT and FREAD are always performed in the SERIAL (non-overlapped) processing mode, regardless of whether overlapped processing is currently in effect. Overlapped processing for other statements is not affected by the presence of FPRINT or FREAD statements.

Previewing a Data Item

One requirement for any READ#, be it serial or random, is that the data types correspond between the data which is stored and the variable into which it is being read. On some occasions, you may not know in advance what the data type of an item might be. In such cases, you will want to find out the data type before doing the READ# and then you will want to select the variable, or variables, accordingly.

This situation can be met with the *TYP function*, which is a numeric function that tests a data type of an item without moving the file pointer from that item. Hence, it is possible to use the TYP function to determine the data type of an item, and then immediately do a READ# to get the item itself.

The function has the form –

TYP (file number)

where **file number** is a numeric expression. The *absolute value* of this expression is the number assigned to the file being read, and must be in the range 1 through 10.

If **file number** is positive, then the function causes the file pointer to remain set to the next data item in the file ignoring all EORs. It returns the type of this next item, and this will be the datum which will be read upon the next serial READ#.

If **file number** is negative, then the function allows the file pointer to remain where it is but it returns the type of the item, even if it is an EOR (type 4). If the type returned is an EOR (4) and then a *serial* READ# is attempted, be wary that the next data item to be read will *not* be an EOR, but will be the next datum *following* the EOR.

The following values correspond to the data types –

Value	Data Type
0	Error – ROM missing or file pointer lost
1	Full-precision number
2	String
3	End-of-file mark
4	End-of-record mark
5	Integer
6	Short-precision number
7	(unused)
8	Partial string – beginning part
9	Partial string – middle part
10	Partial string – last part

The TYP function is a numeric function returning a numeric result. Therefore, it can be used in numeric expressions the same as may any other numeric function. It is an unusual function, though, in that it requires an access to the I/O system. If you do use it in an expression, and should something go wrong in its attempt to get a value (say, the file was inadvertently not assigned), then you will get an error causing the entire numeric expression to abort.

Since the TYP function requires an access to the I/O system, it *cannot* be used in output statements, such as PRINT# or MAT PRINT#. This is to prevent a possible "deadlock" situation where the system would be trying to read (to fulfill the TYP) and write (to fulfill the PRINT#) at the same time.

As an example of the use of the TYP function, suppose you have a serial-access file (assigned to #1), which was written as logical records, each beginning with an integer data item. Suppose further that this item is the only integer in each record and the records otherwise are composed of any number of types of data items. The following program sequence would position the file pointer at the *tenth* logical record –

```
1     INTEGER I
2     SHORT S
3     DIM File$[10]
 .
 .
 .
200   ASSIGN #1 TO File$
210   FOR J=1 TO 10
220   ON TYP(1) GOTO 230,240,250,250,300,260,250,240,240,240
230   READ #1;F
231   PRINT "FULL";F
235   GOTO 220
240   READ #1;A$
245   GOTO 220
250   DISP "ERROR IN POSITION ATTEMPT"
255   STOP
260   READ #1;S
265   GOTO 220
300   READ #1;I
310   NEXT J
```

User-Controlled End-of-File

Upon the creation of any data file, every defined record has an EOF mark placed into it at the beginning of the record. These marks are written over as you print to the defined records. Thus, when you are using a file for the first time, after you complete your serial print statements, there is always at least one EOF mark following your data to indicate the file is complete, until you actually fill the entire file with data.

However, if you are re-using a file (one that has been previously written to in either a serial or random fashion) then where you finish printing your data, there may be some "old" data remaining and no EOF. This could cause difficulties with future uses of the file – trying to determine where the new data leaves off and the old data begins.

To overcome this difficulty, the END data-type was established. By placing the word "END" following the data list in a PRINT# statement (or all by itself), you write a single EOF mark to the file at this point in the file, replacing the EOR mark which is usually written by the PRINT# statement. This is the same kind of EOF written by the CREATE statement, and thus you can use it to detect the end of your data with ON END statements.

Use of the END data type might look like this –

```
100  PRINT #1;A,B,C,D$,END
```

or, it might be all alone –

```
150  PRINT #1;END
```

In the first instance, the END must be the last item, following all other data items in the list.

Notes

Chapter 3

Storage Management

This chapter deals with the efficient utilization of the storage capacity of your mass storage media. The creation, elimination, compaction, and selection of files is discussed along with considerations involving the choice of random and serial access modes for a particular application.

Fundamentals

Storage management involves the usage of certain techniques and tools to control the waste and overhead involved with the storage and retrieval of data in mass storage. Most of the ideas here can be used in many of the applications discussed in later chapters. None of the applications, as will be seen, are mutually exclusive of the others, and you may use them to custom-tailor your mass storage program to a particular need.

Selecting Record Size

You have already been introduced to one statement which can be used in Storage Management – the CREATE statement (pages 29-30). In it, you select a size for a file and define the size of the records in it. The actual size of a defined record which you select should be one which meets the needs of your application. If you are going to use a file as the target of serial access prints and reads, then the size of the defined record is not important. For that reason, it is best to let the record size default (by omitting the **record size** option from the statement). This will maximize the efficiency of the system's mass storage routines. However, while the defined-record size may not be so important, the file size is. It is necessary that your file be large enough to hold the data you will be writing to it *in toto* (plus any bytes which may have been wasted in the records).

When you are planning your record, you may also consider the size of the individual items going into them. Occasionally it is possible to save storage by selecting a different data type. For example, if you are storing 1,000 full-precision numbers, it will take 8,000 bytes of storage. But if you know in advance that the actual values will be whole numbers within range of the INTEGER data type, then you could reasonably change the type to INTEGER and the total storage requirement would be cut in half to 4,000 bytes.

Overflowing Files

If a file size was selected that is too small to handle all of your data, you will have to create another file to handle the overflow. Then you will have to use a method of switching from the full file to the new one. For example, here is one method which utilizes "naming conventions" for files to accomplish this purpose for a particular file –

```
100  PRINT #1;A,B,C,D$,END
10   !
20   !  Select initial file name
30   !
40   DIM Device$[3],File$[5]
50   File$="DATA"
60   Device$=":F"
70   Records=10
80   Counter=1
90  Filename$=File$&VAL$(Counter)&Device$
100  CREATE Filename$,Records          ! Create and assign
110  ASSIGN #1 TO Filename$            ! The first file
120  ON END #1 GOSUB Newfile           ! When file's end is
130 !                                  ! reached, create another
140 !  Program follows
150 !
 .
 .
 .
500  PRINT #1;...
 .
 .
 .
1000 !
1010 ! File-creation routine follows
1020 ! Creates files by incrementing a counter and using it
1030 ! as part of the file
1040 Newfile:  Counter=Counter+1
1050 Filename$=File$&VAL$(Counter)&Device$
1060 CREATE Filename$,Records      ! Defaults to
1070 ASSIGN #1 TO Filename$        ! physical-record size
1080 PRINT #1;...
1090 RETURN
```

In this example, a file called "DATA1" is initially created and then printed serially by repetitions of statement 500. Finally, should the file become full, the ON END will cause "Newfile" to be performed which creates a file called "DATA2", assigns it to #1, and prints the information to *it* instead of the first file. This could continue until the program is finished, creating files each time the previous one is filled. Upon completion, there will be stored on device ":F" (the flexible disk drive), one or more files called "DATA...." After the program is finished a CAT "DATA:F" would reveal the directory data on the files created.

The ON END statement that was used here is one of the tools of both storage management and mass storage processing in general. It is used to indicate an action to be taken whenever an end-of-file (EOF) condition is encountered during record I/O on the file number indicated, or an end-of-record (EOR) condition is encountered during random-access record I/O. The statement may take any of three forms –

ON END # file number CALL subprogram name
ON END # file number GOSUB line identifier
ON END # file number GOTO line identifier

line identifier may be a line number in the program, or a label.

subprogram name is the name of a currently existing subprogram.

ON END statements are executable statements. There may be any number of ON END statements in a program. Some may be for different file numbers; some may be for the same file number. Should an EOF (or an EOR in random access) be encountered on such a file, the action taken will be that indicated by the last ON END statement executed for that file number. Executing an ON END statement overrides a previous ON END for the same file number; it also cancels OVERLAP for that file (see pages 66-67).

In production applications, where efficient performance is desirable, the OVERLAP-cancelling effect of ON END can be detrimental. To avoid a loss of efficiency in such cases, it is better that you should avoid the ON END statement and use programming methods which enable you to keep track of where you are in the data (and in the file). Record-counting (or data item-counting) is among the best of such methods.

You can turn off this END-testing condition with the OFF END statement. It has the form –

OFF END # file number

Until an ON END declaration is overridden with another one for the same file number, or an OFF END is executed for that file number, it remains in effect – thus it only needs to be executed once.

Copying Files

Sometimes there is a need for an exact duplicate of a file. This often occurs when you might need (or want) a backup for a critical file, or to transfer a file from one medium to another. The statement to use for this purpose is –

COPY source-file specifier TO destination-file specifier

Execution of this statement creates the destination file (so there must not already be a file with this name on the destination medium) and copies the records of the first file to the newly-created one. After the copy is complete, the second file will be identical to the first – same file type, record size, number of records, and contents. Both files will then exist.

Any kind of file may be copied with this statement, not just those of the DATA-type. See also "Protecting a File" below (page 52) when copying a file which has a protect code.

Purging Files

Occasionally it is desirable to remove a file from the mass storage medium. Obsolete files, temporary backup files, and botched data files are some of the types of files that can "collect" on a medium, cluttering the directory and wasting useful storage space. To get rid of the unwanted ones, the statement to use is –

PURGE file specifier

and the file is permanently removed from the directory.

As an example of the utility of the COPY and PURGE statements, the following catalog of a tape shows a rather full storage medium –

```
NAME   PRO TYPE REC/FILE   BYTES/REC    ADDRESS
T15           2

DATA1      DATA      68       256          5
DATA2      DATA      39       256         73
DATA3      DATA      42       256        112
DATA5      DATA      75       256        205
DATA7      DATA      60       256        331
DATA9      DATA      89       256        430
DATA10     DATA      82       256        519
DATA11     DATA      88       256        601
DATA12     DATA      98       256        689
DATA13     DATA      41       256        787
```

While quite a few gaps exist, amounting to considerable storage space on the entire tape, there is not enough contiguous space on the tape to be able to add another file 100 records long (256 bytes each). To add such a file, there is a need for 100 records *in one location* on the tape. Since that doesn't exist, the tape should be "re-packed" –

```
10   MASS STORAGE IS ":T"
20   ON ERROR GOTO Error
30   Temp$="TEMP:F"
40     FOR I=1 TO 73
50     File$="DATA"&VAL$(I)
60     COPY File$ TO Temp$ ! Make temporary copy of the file
70     PURGE File$         ! Get rid of original
80     COPY Temp$ TO File$ ! Rewrite file, taking advantage of gaps
90     PURGE Temp$
100    NEXT I
110  CAT
120  STOP
130 Error: IF ERRN=56 THEN GOTO 100  ! A mere gap in the numbering?
140  DISP ERRM$                      ! Or something more serious?
150  END
```

This example relies upon a naming convention (i.e., DATA...) for the files. Successively, starting with the first file on the tape, each file is read, temporarily stored on another device, and then purged. This frees up the space it previously occupied, along with any "gaps" in storage available before or after where it was stored. Since each file is stored at the first available spot where it can be fit, the files will be written one immediately following the other. Progressively, then, the gaps between files will be collected together so that, by the end, all the available storage area is at the end of the tape – and all in one spot.

In this example, this would mean that there will be sufficient room to write the new 100-record file. The re-organized catalog would look like –

```
NAME   PRO TYPE REC/FILE  BYTES/REC   ADDRESS
T15          2

DATA1      DATA      68       256          5
DATA2      DATA      39       256         73
DATA3      DATA      42       256        112
DATA5      DATA      75       256        154
DATA7      DATA      60       256        229
DATA9      DATA      89       256        289
DATA10     DATA      82       256        378
DATA11     DATA      88       256        460
DATA12     DATA      98       256        548
DATA13     DATA      41       256        646
```

Special Operations

Protecting a File

The purging capability is a strong one. An inadvertent use of the PURGE statement can be disastrous under some circumstances, since purging a file is permanent and the file is irretrievable.

If you do not have backup files for your critical data, accidental erasure can lose that data permanently. Backup files, of course, are one solution, but they double your storage requirements. Whether or not you employ backup files, there are additional methods to protect a file which you do not want accidentally purged.

One method is to "protect" it is with a protect code. This code, which is similar to a password, enables you to establish write-protection to an individual file without having to protect the entire medium. To assign a protect code to a file, use the statement –

PROTECT file specifier , protect code

protect code is any string expression containing any character except the quote-mark and the blank. It may have any length, except 0 (i.e., it can't be a null-string[1]), but only the first six characters will actually be saved as the code. For tape cartridges, the directory does not retain the protect code itself, and only notes the fact that you have protected the file. For all other mass storage devices, the protect code itself is kept with the file description in the directory.

To purge a protected file, it is necessary to add its protect code to the PURGE statement. This is the same protect code which was used in the PROTECT statement for the file (except with tape cartridges, where *any* protect code will do). Thus the PURGE statement for a protected file must be –

PURGE file specifier , protect code

A protected file is also protected against all potential attempts to write-access the file without the **protect code**. Thus, PRINT# statements to the file will not work, unless the ASSIGN to the file was made with the proper protect code (see page 31). However, a GET statement, all other things being equal, will be permitted, since it only *reads* the file.

It is also necessary to add a protect code when attempting to copy a protected file. The COPY statement would then look like –

COPY source-file specifier TO destination-file specifier , protect code

The new file created by the COPY will have the same protect code as the old file.

[1] A null-string will be interpreted as "no protection".

If you are copying a file from a tape unit to a non-tape unit, you may inadvertently create a protect code for the new file. While a protected file on a tape can be copied using *any* protect code in the COPY statement, the new file will be protected with the protect code which you used in the COPY. Be certain, in such instances, that you make note of the protect code or you will not be able to access the new file in the future. For example, with –

```
PROTECT "Employ:T15","Pay"
COPY "Employ:T15" TO "Empbak:F8","Temp"
```

the file "Employ" on T15 can be accessed with any protect code (the COPY was an example), but the "Empbak" file on F8 can only be accessed with the protect code "Temp".

If you have protected a BDAT file, then all FPRINT and FREAD statements must have a protect code included. The statements in this case would appear as –

```
FPRINT file specifier , protect code , array identiier
FREAD file specifier , protect code , array identifier
```

Renaming Files

Should you want only to change the name of a file as it is kept in the directory, the statement to use is –

```
RENAME old-file specifier TO new-file name [, protect code]
```

old-file specifier is a file specifier, but **new-file name** is a file name only. The **protect code**, of course, is a string expression for the proper protect code for the file. The **protect code** will not *add* protection to a file if it does not already have it; it merely allows you to access the file for renaming if it is protected.

Execution from the Keyboard

The COPY, PURGE, PROTECT, and RENAME statements may all be executed from the keyboard.

Notes

Chapter 4

Data Transfers

Along with the capabilities that mass storage gives you, there is the problem of how to make data operations as efficient and reliable as possible. Associated with this is finding ways to improve the overall reliability and security of your data base, and finding efficient methods for passing data from one program to another.

Among the techniques available to you on the System 45 are buffering; using arrays, backup files, and consistent formatting; and employing the overlapped processing capability.

Buffering

Device Buffering

Doing many PRINT# and READ# operations to mass storage can cause the speed of program execution to slow noticeably.

To minimize the impact of mass storage operations on processing speed, the concept of device "buffering" is employed by the System 45. For every mass storage select code used by a program, there is a 256-byte buffer used in I/O operations to a device on that select code. If there is more than one device on a select code, then they all share the same buffer. Use of these buffers reduces the number of physical accesses necessary to transfer information between a mass storage device and memory. This in turn reduces the time spent by an executing program waiting for an I/O operation to a file to be completed.

The following example demonstrates the capability –

```
100  ASSIGN #1 TO "DATA"
110  FOR I=1 TO 100
120  PRINT #1;A(I)
130  NEXT I
```

Each time the PRINT# statement in the above is executed, the value contained in A(I) will be written to the *buffer* for the device, and not directly to the device itself. Since A(I) is a full-precision number, eight bytes will be written to this buffer each time PRINT# is executed. After 32 iterations all 256 bytes of the buffer will be filled. Then the entire contents of the buffer will be physically written to the device, the buffer will be erased, and it will start to fill all over again. This is called "flushing" the buffer. Each time the buffer fills, it will automatically flush to the device. Thus, instead of making 32 physical accesses to the file in this case, the program needs to make only one – a significant saving.

Buffer flushing will also occur whenever a STOP or END statement is encountered, and whenever PAUSE or STOP is entered from the keyboard. The buffer for a particular file will also be flushed whenever access is made to:

- a different physical record within the same file;
- a different file on the same device;
- a different device on the same select code.

In addition to a savings in execution time with the use of buffers, there is also a reduction in wear of the medium when used with tapes and flexible disks.

With a READ# operation, the effect is similar. A full buffer's worth of information is read into the device buffer first, then successive read statements will take information from the buffer instead of directly from the device itself. When the information in the buffer is exhausted and more READ# statements are executed, another 256 bytes will be physically read into the buffer and information is then taken from the newly-buffered data.

The device buffers are an integral part of the I/O system of the System 45. Each mass storage device will access data through its buffer. If you are both reading and writing files to the same device in the same program, the buffer advantages could be lost, depending upon how you organize the program's mass storage operations. For example, a program segment such as –

```
100  ASSIGN #1 TO "DATA"
110  ASSIGN #2 TO "NEW"
120  FOR I=1 TO 100
130  READ #1;A(I)
140  PRINT #2;B(I)
150  NEXT I
```

would have a vastly improved execution performance if the statements were reorganized so that access was made to only one file at a time –

```
100  ASSIGN #1 TO "DATA"
110  ASSIGN #2 TO "NEW"
120  FOR I=1 TO 100
130  READ #1;A(I)
132  NEXT I
134  FOR I=1 TO 100
140  PRINT #2;B(I)
150  NEXT I
```

Since READ# operations use the device buffer in a different way than do PRINT# operations, the buffer for the device containing "DATA" and "NEW" must be flushed whenever there is a change from reading to printing, and vice versa. Also, since we are reading from one file and writing to another, there are different parts of the medium to be accessed.

Therefore, in the first program segment, the buffer is flushed each time there is a PRINT#2 following a READ#1 – which is every time the loop is executed. Also, the buffer is flushed each time there is a READ#1 following a PRINT#2 – again, every time the loop is executed. The value of having a buffer has been lost.

Alternatively, in the second program segment, all of the reads (and accesses) to file#1 take place together – allowing a full utilization of the buffering capability. After all the reads are done, then all the prints take place to file#2, and again it takes full advantage of the device buffer.

In addition, the first program segment requires considerable motion on the part of the mass storage device as it continually has to reposition itself between the two files. If the files are physically far apart, there could be considerable time spent just moving from one part of the medium to another. The second program segment minimizes this difficulty by causing all accesses to each file to be performed one after another, rather than alternating between the files. The result, in this instance, is that system repositions the medium only once.

Overriding the Device Buffer

The presence of a buffer on every device is highly valuable in applications where a great deal of reading and writing is being done to mass storage devices. However, when a program is more "compute-bound" than "I/O bound" and only occasionally writes a value out to a mass storage device, the buffering capability can have a pitfall.

In cases of power failure, device malfunction, etc., data which has been accumulating in a buffer may be lost, though they theoretically have been "written" to the device. At most, only 256 bytes could be so jeopardized per device, but in some applications, the risk may not be an

The BUFFER statement sets up an *additional* I/O buffer associated with a *file number* and not with the device the file happens to be on. It works in the same way as does the device buffer, but it is not subject to some of the same limitations as is the device buffer. Primarily, it is not flushed when operations take place to other files on the same device.

When you allocate a file buffer to an assigned file, with the following statement –

BUFFER # file number

you set up a separate 256-byte buffer for all READ# and PRINT# operations to that *file number*. It acts, in this regard, the same for the file as the device buffer does for the device. When it becomes full, and is ready to be flushed, it flushes to the device, *only then* causing the device to take notice that READ# and PRINT# operations to its files have been going on. Hence, you can be reading and printing to many different files, but you involve the device only when a file buffer is flushed.

If the following two statements are added to the example above –

```
105  BUFFER # 1
115  BUFFER # 2
```

we would achieve the same kind of savings we would expect to achieve if the device buffers could have been used effectively.

A file buffer is flushed whenever its file number is reassigned through an ASSIGN statement (see pages 31-33). The buffer is flushed before the actual reassignment is made so that buffered information is transferred to the *old* file and not the new.

Every file which is assigned a file number through an ASSIGN statement may have a buffer allocated to it through the BUFFER statement. The space required in memory to create these buffers (256 bytes for the buffer plus a 13-byte overhead) is taken from your usable memory. Thus, if you have an application which is tight on usable memory, you may want to be selective on the use of buffers.

As with the device buffer, a file buffer will be flushed if a PAUSE, STOP, or END statement is executed (from a program), or if PAUSE is entered from the keyboard.

File buffering is only really effective when such buffers are established for *every* file on a particular device.

Conflict Between CHECK READ and BUFFER

The individual missions of the CHECK READ and BUFFER statements are in conflict over the role of "immediate-record".

If CHECK READ is in effect for a file, one of its effects is supposed to be to force an immediate device buffer flush after every PRINT#. But if BUFFER is also in effect for a file, the intent is that device buffer flushing be suppressed until a file buffer is filled.

To reconcile this conflict, the BUFFER statement is presumed to predominate over the CHECK READ. If a BUFFER statement for a particular file is in effect, a CHECK READ to that file will still perform its "read-after-write" function, but only after the file buffer has been filled and flushed. It will not flush the device buffer immediately as it would if the BUFFER statement were not in effect.

Using Arrays as Buffers

Still another way to get a "buffering" effect is through the use of arrays. By reading or printing arrays as a *unit* rather than as individual elements, you get the maximum utilization of your device buffers, without the need of a file buffer.

The PRINT# and READ# statements using an array identifier (see the Operating and Programming Manual), or MAT PRINT#, can be used to transfer data in an array. For example, with an array of 1,000 elements, the following program segment –

```
470  FOR I=1 TO 1000
480  A(I)=RND
490  NEXT I
500  PRINT #1;A(*)
```

is faster (with or without a BUFFER statement), than is –

```
470  FOR I=1 TO 1000
480  A(I)=RND
490  PRINT #1;A(I)
500  NEXT I
```

and because of the way the System 45 language is structured, faster still are the FPRINT and FREAD statements for non-tape media. For details on these statements, see pages 39-42.

Using an array as a buffer (this only works for data which are all of the same data type), you can achieve buffers of greater lengths than 256 bytes (which is all that the BUFFER statement will give you), and you can avoid the conflict between CHECK READ and BUFFER. This approach

permits you to get the immediate-write facility of the CHECK READ, but avoiding the cancelling effect of a BUFFER. You can still get the buffering capability by storing things in an array then writing the *array* out all at once. For example –

```
30   CHECK READ #1
 .
 .
 .
190  PRINT #1;Z
200  FOR I=1 TO 100
 .
 .
 .    [calculate C]
 .
 .
390  A(I)=C
400  NEXT I
410  MAT PRINT #1;A
420  INPUT Z
430  GOTO 190
```

In this example, you get the immediate-write facility for Z, thus assuring that its values are physically written to the device. You could have printed each C individually as well, but by doing so, given there are so many of them, you would slow down the execution speed because of the CHECK READ. Instead, by storing them in an array and executing just one PRINT#, you speed up the execution appreciably. Of course, you also get the verifying "read-after-write" effect for both PRINT# statements.

Advanced Techniques

Passing Data Between Programs

With the advent of modularized and structured programming, the use of mass storage as a medium for passing data between modules and programs is increasing. Also, as data base management increases in popularity and necessity, this use as a data-passer should increase. In all such applications, the by-words are "consistent formatting".

When passing data between programs, it is necessary that the program which prints the data, and the program which reads the data agree on three things –

- Data type of each data item.
- Number of data items per logical record.
- Application, or "meaning", of each data item.

Data transfers between programs rely primarily upon the system design that calls for them. Hence, consistent system designs, thorough program and system documentation, and well-checked programs (modules), are the best tools for reliably passing data. There are any number of particular implementations, however, which can ease the difficulty of achieving a consistent format for passed data –

1. Use the TYP function (pages 42-44) to assure that items are of the proper data type.
2. Use random access mode whenever feasible. This permits you to control better your position within your data. When using serial files, also use the TYP function to ensure that you start reading a logical record from its beginning. Make your logical records and defined records the same length.
3. Use mnemonic variable names and, if at all possible, make them consistent from module to module. When you print something to a file, you only print the data and not the variable name where the data was being held, so you may print something using one variable name and read it with another variable name.

It is most wise during any attempt at consistent formatting that each data type exactly correspond between variable and datum being read. This is strictly true when strings are involved. It is not possible to read a string datum into a non-string (i.e., numeric) variable. Similarly, it is not possible to read a numeric value into a string variable. Thus, you must be constantly aware whether a string or non-string is involved in your I/O operation or else an error may occur.

While it is advisable that the data types correspond among the numeric data types as well (full, short, and integer), it is not as strictly necessary that there be a correspondence between the variable types and the data. In the *numeric* case only, a difference in data type between datum and variable will cause the data to be converted to the type of the variable. This can alleviate some of the headaches involved in consistent formatting, but by no means all. You will have to watch out for potential "overflows" and "underflows" in the data, and the potential loss of precision due to truncation.

For example, trying to store either of the following numbers –

5.555 E-70
3.143 E85

into a short-precision variable will cause an underflow and overflow respectively as the system tries to convert them to the proper storage representation.

In a READ# statement, conversion of a data item which is a full- or short-precision value into an integer variable involves *truncation* of the non-integer portion of the value. This is different from the usual assignment (LET) operations. Whereas the *assignment* of the value 1.8 to an integer-type variable, such as –

Integer=1.8

would result in 2 being stored, the *reading* of the value 1.8 into an integer-type variable, such as –

READ#1;Integer

would result in the value 1 actually being stored.

The same kind of truncation occurs in the low-order digits in the conversion of a full-precision data item into a short-precision variable.

As a demonstration of data passing, suppose you had an order-generating program, which said in part –

```
100  DIM Customer$[24]
110  SHORT Supply
120  INTEGER Order
130  CREATE "ORDERS:F",100,42
140  FOR I=1 TO 100
 .
 .
 .
550  PRINT #1,I;Customer$,Supply,Order
560  NEXT I
```

then you could use the file created by this program in another shipping program which says in part –

```
100  DIM Customer$[24]
110  SHORT On_hand
120  INTEGER Shipping
130  ASSIGN #1 TO "ORDERS:F"
140  FOR I=1 TO 100
 .
 .
 .
700  READ #1,I;Customer$,On_hand,Shipping
 .
 .
 .
900  NEXT I
```

Backup Files

For reliability, it is recommended that you maintain "backup" files for your critical data and programs. Usually, it is best that these backups be on different media, so if anything happens to one medium you will still have the other to use. Of course, this requires extra machine time to accomplish the copying.

Backups are easy to create through the COPY statement (see page 50 for details). If you are trying to copy a different medium of the *same type*, but have only one drive of that type, you may have to use two COPY statements – one to an intermediate file on the tape cartridge unit (provided the file isn't too large for it to hold), and another to copy the intermediate file back to the original drive after you've switched media. For example, the following would create a copy in such a fashion using a single flexible disk drive –

```
900  Temp$="######:T"
1000 File$=":GEORGE:F"
1010 COPY File$ TO Temp$
1020 DISP "Please insert backup floppy"
1030 PAUSE
1040 COPY Temp$ TO File$
1050 DISP "Please re-insert primary floppy"
1060 PAUSE
```

Don't forget, in such an application, to purge the temporary file to avoid cluttering up your media with unneeded copies.

It is feasible to use the COPY statement with only one drive, but the process is slower than it might be if two drives are involved, especially with the tape cartridge unit.

Overlapped I/O

Another way to speed up the throughput of your programs is to take advantage of the overlapped-processing capability of the System 45.

It is quite possible for the System 45 to be working on transferring data even as it is busy calculating and making decisions.

In the following program –

```
100  FOR I=0 TO 100 STEP .1
110  A=SIN(I)
120  B=COS(I)
130  C=TAN(I)
140  PRINT A;B;C
150  NEXT I
160  STOP
```

the normal execution sequence (called the "serial" mode) is to calculate A, then B, then C, and then print all three and loop back and do it all again. But in an "overlapped" mode, the three calculations are done and the results turned over for outputting. Then, even while the System 45 outputs these values, it goes back and does the next three calculations, and so on. So actually, the System 45 is doing two things at once.

This feature can be used to advantage in mass storage applications by placing as many non-I/O statements between your PRINT# and READ# statements as the logic of your program will permit. In general, a greater separation between I/O statements maximizes the value of this feature.

You may enable this characteristic by executing the statement

```
OVERLAP
```

from either the keyboard or a program. To go back to the normal mode of doing things, the statement

```
SERIAL
```

should be used. OVERLAP can also be cancelled by SCRATCH A and reset. SERIAL is the power-on mode.

Another situation will also disable the OVERLAP mode. If *an* ON END statement is in effect, OVERLAP will necessarily be disabled for that file. This effect occurs because of the intended purpose of the ON END statement. The ON END intends that some branch – and alternate execution sequence – should take place when the END condition is encountered on a file. If you are in the overlapped mode, however, other statements than are intended might be executed before the end condition is recognized. Thus, enabling an ON END will partially cancel the effect of an OVERLAP. After executing an OFF END for a file, you will automatically go back to overlapped processing. For a discussion of the effects on programming methods, see Overflowing Files in Chapter 3 (pages 48-49).

The ASSIGN, FPRINT, and FREAD statements are always executed in a non-overlapped (SERIAL) mode, regardless of the OVERLAP status. The OVERLAP mode also cancels any ON ERROR trapping for CHECK READ.

A REDIM statement for an array should not follow, in close proximity, an FPRINT statement using the same array while processing is in the OVERLAP mode. For details on this restriction, consult the cautionary note on page 42.

To get back from storage a program which has been stored by a STORE or RE-STORE statement, the statement to use is–

LOAD file specifier

This statement will load the file into memory, and it is then available for your use as would be the same program entered from the keyboard.

If you want to immediately execute a program after loading it, you can either use the LOAD statement as above and then press the RUN key after it is loaded, or you may add an execution line as a parameter to the LOAD statement, such as –

LOAD file specifier , execution-line identifier

The **execution-line identifier** is a line identifier for where you want the program to begin executing. Normally, this line would be the first line in the program, but there is no requirement that this be chosen. You may want to start it somewhere else.

When a program is loaded, it destroys the previous program and variables stored in the memory (except for variables stored in common – COM) and replaces them with the version stored in the indicated file.

Another way to keep a program on mass storage is to save its source-code version. By using the statement –

SAVE file specifier

you take the current program (which is stored in memory in quasi-compiled form) and reverse-compile it into strings, one for each line of code – the same as if you were listing the program. Then these strings are stored, in order, in the file indicated (in serial fashion) into a DATA-type file. You can use this file as a data file if you want. It is a serial-access file with up to 160-byte strings forming the logical records – one logical record for each program line.

You don't have to store the entire program if you don't want to. You can select the lines you want to save by using one of the following versions of the statement –

SAVE file specifier , first-line identifier
SAVE file specifier , first-line identifier , final-line identifier

In the first case, only that part of the program starting with the **first-line identifier** will be saved. In the second case, only that part of the program between and including the two lines indicated will be saved. **final-line identifier** must be greater than **first-line identifier**, if both are present. If the line(s) indicated do not exist, the line with the next-highest line number will be used in each case.

You may also replace a previous version of a program with the same file name on the medium with the RE-SAVE statement. This statement has the same properties and parameters as the SAVE statement –

RE-SAVE file specifier [, protect code] [, first-line identifier [, final-line identifier]]

The way to get a saved source version of a file from a medium is to execute the statement

GET file specifier

This statement reads the DATA-type file indicated and compiles the strings (lines) as it finds them. If any of them (for some reason) do not compile correctly, it makes comment lines out of them by placing an exclamation point ( ) after their line numbers. The program, after this is finished, is compiled and ready for your use, the same as if it had been entered from the keyboard.

Unlike LOAD, the GET statement does not necessarily erase all of the program which is already in memory. Rather, by adding a parameter to the statement –

GET file specifier , line identifier

it keeps all line numbers which are *less* than the **line identifier** in the original program. For example, if there were a file which started with the following lines –

```
40   FOR I=1 TO 100
50   PRINT #1;VAL$(I)&" PRINT RND"
60   NEXT I
```

and you executed a GET on this file with a **line identifier** of 40, then a program already in the memory with the lines –

```
10   DIM Line$(10)
20   CREATE "RANDOM",100,20
30   ASSIGN #1 TO "RANDOM"
40   ! THE FOLLOWING IS A TEST ROUTINE ONLY
```

would become –

```
10   DIM Line$(10)
20   CREATE "RANDOM",100,20
30   ASSIGN #1 TO "RANDOM"
40   FOR I=1 TO 100
50   PRINT #1;VAL$(I)&" PRINT RND"
60   NEXT I
```

The remainder of the program would be that from the file. All lines of the original program after line 30 will have been replaced or deleted by lines in the file program.

This feature accomplishes two purposes. First, it preserves all lines of the program in memory which have line numbers less than **line identifier**. All program lines in memory which have line numbers greater than **line identifier** are deleted. If **line identifier** is a label, then this effect uses the line number of the program line which contains that label.

Second, it *renumbers* the file it retrieves, starting with **line identifier** and preserving the line-number spacing between lines. For example, if you GET a file which is numbered 10, 11, 12, 20, and 30, but the **line identifier** is 50, then the lines will be stored in memory numbered as 50, 51, 52, 60, and 70. If **line identifier** is a label, then this effect uses the line number of the program line previously in memory which contained that label.

If you had executed the GET in the previous example with a line identifier of 50 on the above file, the result would have been –

```
10   DIM Line$(10)
20   CREATE "RANDOM",100,20
30   ASSIGN #1 TO "RANDOM"
40   ! THE FOLLOWING IS A TEST ROUTINE ONLY
50   FOR I=1 TO 100
60   PRINT #1;VAL$(I)&" PRINT RND"
70   NEXT I
```

You can also select a line at which you might wish to begin execution of the newly-arranged program by adding still another parameter –

GET file specifier , line identifier , execution-line identifier

Immediately upon loading the program, execution of the program will commence with **execution-line identifier**. In general, if this parameter is omitted, execution will begin with the **line identifier** (exact rules can be found on the next page).

Using a GET also causes all values stored in variables to be destroyed, except for those variables in common (COM statement). If this is an undesirable effect, the LINK command should be used instead. This statement has the same parameters as the GET –

LINK file specifier [, line identifier [, execution-line identifier]]

and has the same effects as GET, except that the value of *all* variables are retained.

This means the GET statement is a way for passing information (via the COM statement) between the programs. (Using LINK instead of GET will allow information to be passed outside of common.) Since the LOAD statement (as with GET) does not reset the value of variables in common, it is also possible to pass information using the LOAD statement. In fact, using the LOAD in preference to GET will cause significant savings in execution time associated with

retrieving the program. Improvements in performance using LOAD over LINK can be as much as 3:1 with tapes, 5:1 with flexible disks, and 14:1 with hard disks, depending upon the relative transfer rates of the device involved.

Using a GET statement in a program without an execution-line identifier has a different effect than the same statement executed from the keyboard. When executing such a GET from the keyboard, after the program is retrieved, control returns to you. But, when executing it from a program, a GET automatically begins execution. If you specify an execution line, then execution will commence at the line indicated. If you do not specify an execution line, then execution will proceed according to the following rules –

- If the **line identifier** is less than or equal to the current line which originally contained the GET (as stored in memory), then execution proceeds with **line identifier**. For example –

```
50   GET "Backup",40
```

  execution will immediately begin with 40 (since it is less than 50) after "Backup" is retrieved.

- If the **line identifier** is greater than the current line which originally contained the GET (as stored in memory), then execution proceeds with the first line of the program which follows the GET. This next line may be one of the first lines of the retrieved program, or a line remaining from the original program. For example –

```
50   GET "Backup",100
60   Flag=1
```

  "Backup" will be loaded beginning at 100, and then execution continues with line 60.

NOTE

When using whole programs without the need to renumber, etc., use STORE and LOAD instead of SAVE and GET. The former commands execute with greater speeds than do the latter.

A third way to get a program onto mass storage is to create it from a program, create a DATA-type file, and store the program lines as strings into the file. Such a program can then be the object of a GET statement as above.

As an example of creating a program, try the following –

```
100  DIM Line$[24]
110  CREATE "RANDOM",100,20
120  ASSIGN #1 TO "RANDOM"
130  FOR I=1 TO 100
140  PRINT #1;VAL$(I)&" PRINT RND"
150  NEXT I
160  ASSIGN * TO #1
170  GET "RANDOM"
180  END
```

This program will create the file and store in it 100 generated strings which themselves make up another program. The GET statement will then fetch this program. When a GET is executed from a program, the new program just retrieved is automatically executed, so this program will, in the end, print 100 random numbers as well as changing the original program to 100 lines of "PRINT RND".

If you use this feature to get and run a number of programs in succession, or to add a number of subprograms, you should be aware that any errors associated with the GET (such as error 80 or 81, for example), cannot be trapped with an ON ERROR. Thus, this operation can represent a potential area where your ON ERROR planning will not work.

Storing Key Definitions

If you have a program stored on mass storage which makes use of certain special function key definitions, you can gain better control over your user's operating environment by storing the special function key definitions (not to be confused with the ON KEY declarations which you may have in your program) and retrieving them from the program.

To keep the current set of key definitions and put them in a file for later use, the statement to use is

STORE KEY file specifier

The file created by this statement will be a KEYS-type file.

Then to retrieve the definitions, execute the statement

LOAD KEY file specifier

and the current key definitions will be destroyed and the ones stored in **file specifier** will be loaded in their stead.

This operation can be done from the keyboard, but is most usefully employed in a program. For example, if the program –

```
10   LOAD KEY "Select"
20   DISP "Press appropriate key for routine selection"
30   ON Selection GOSUB ...
 .
 .
 .
```

were executed, and the file "Select" had key definitions in it which appropriately set the value for "Selection", then the appropriate subroutine would have been chosen for execution.

Special Situations

Loading Binary Programs

If you have acquired a number of binary programs, they can be loaded with the statement –

LOAD BIN file specifier

The routines in **file specifier** will be loaded *in addition* to any other binary routines, and a program, which may be present.

If you want to store the binary routines presently held in memory, the statement is to use –

STORE BIN file specifier

The statement will cause all of the binary routines currently in memory to be stored together in **file specifier** as a BPRG-type file.

Memory Snapshots

The entire contents of memory – stacks, buffers, display, program, variables – can be stored in one file as a "machine state" file, or "memory" file. The statement to accomplish this is –

STORE ALL file specifier

When this statement is executed, the current contents of memory will be transferred to the file indicated. Such an "ALL-type" file can only be retrieved with a LOAD ALL command, which has the form –

LOAD ALL file specifier

When LOAD ALL is executed, the contents of the file are dumped bit-for-bit into the memory and the "machine state" or "snapshot" which the file contains will then *become* the machine state.

If power is lost to a mass storage unit and the unit is not needed by the system, then the loss will not affect operation. If the unit is needed, however, for a PRINT# or READ# operation (including the TYP function), the message –

```
I/O DEVICE TIMEOUT ON SELECT CODE nn
```

will be displayed and the system will pause and periodically query ("poll") the missing (or malfunctioning) unit ("nn" will be the number of the select code where the problem exists). It will continue this way until the unit is brought on-line, at which time the command –

```
READY#nn
```

should be executed from the keyboard. The system will poll the unit one more time and then allow the program to continue.

- **Equipment failure.** Occasionally, because of a failing component, or electrical noise, the computer or one of its mass storage units gets itself into a state from which it can neither operate nor recover. If it is the computer, the most obvious symptom is the "system lockup" (where the machine does not seem to respond to any reasonable keyboard input). If it is one of the devices, the "TIMEOUT" message may be returned, or perhaps one of the system error messages (between 69 and 89) will appear.

- **Cable separation.** The desktop computer may lose communication with external mass storage devices should the cable connecting them malfunction. It will definitely lose contact should the cable become physically separated from either the computer or the device.

- **Media wear.** Because of frequent contact with the read/write mechanisms of a mass storage unit, a medium will begin to wear, causing unreliability for data storage. Since the directory is usually the most frequently accessed area on a medium, it is the most likely to wear out first. When this happens, the alternate directory is accessed and you are given the warning –

```
SPARE DIRECTORY ACCESS
```

Should you ever receive this message, it is a sign that the medium is beginning to wear out, or that the data on the main directory has become garbled and unreliable.

What To Do About Hardware Errors

Should you experience a power loss to any of the units and you are present before power returns, switch off the affected units before power returns. Check power cords and fuses before switching on again.

If a power surge was experienced (or is suspected) before the loss, it would be wise to conduct a system test of all units (see the system test booklet for each device).

If equipment failure is suspected, perform the system test on the device concerned. If the test comes out indicating a problem, or if the device will not work sufficiently to even allow the test to be performed, then call an HP Sales and Service Office (see Appendix F).

Should the interfacing cable become separated, re-insert and attempt to proceed. Should a system test show it to be malfunctioning, replace it with an identical one (and set the same select code), or call for service.

Any device or cable failure, except the computer itself, can be corrected by replacing the malfunctioning unit with one in proper operating order. The medium should be placed into the new unit and processing should be able to proceed where interrupted. It is possible, however, in such circumstances, that failure may have occurred during I/O transfer. If such was the case, you probably lost data.

With medium wear, it is highly advisable to copy all files to another medium. Whenever you receive a "spare directory access" message, the files should be copied immediately. To continue to use the old medium with spare directory accesses is to risk the loss of all data on the medium. Not only may the data read and written to such a medium be potentially unreliable, but should the spare directory itself ever wear out, you will be without a backup and will no longer have any means of accessing *any* of the files on the medium.

Anticipating Hardware Errors

Hardware difficulties are part of the external environment which a *program* cannot control, but that a *programmer* can anticipate.

In the case of power outages to the external units, as well as equipment failures involving those devices and cable separations, program control can be retained through the anticipatory use of the ON ERROR facility, with clever use of the ERRL and ERRN functions (see the Operating and Programming Manual for details on the functions themselves). Making software provisions for attempting to use alternative drives, or for just "dying gracefully", is the best manner of dealing with the unanticipated nature of these problems.

In the case of failure of the desktop computer itself, the problems of recovery can be nearly insurmountable under program control. The best insurance against computer failure of all types is to be liberal in the use of backup files and memory snapshots. The use of manageable checkpoints in your programs, wherein you require human intervention, is another tool particularly useful in spotting the more subtle forms of processor failure, when and if they occur.

In some applications, the particular use of memory snapshots (STORE ALL) for recovery purposes is preferable over alternative methods. This is particularly true when an ON END statement is in effect. Because it is possible for other processing to continue while an I/O operation takes place, precise recovery characteristics may be confusing. But memory snapshots can give an exact picture for recovery purposes.

Error Messages (Hardware)

The following is a list of the system error messaages you may receive should there be a hardware-related problem of some sort. A possible corrective action, or actions, is included in the discussion of the error.

ERROR 1 ROM missing, or configuration error. To operate the System 45, all system ROMs and the Mass Storage ROM must be in place. Perform the system test if the problem persists.

ERROR 57 Similar to ERROR 1 above. To operate an external mass storage device, you need the Mass Storage ROM. Check the ROM drawer to see that the ROM is in place. Perform the system test if the problem persists.

ERROR 65 Incorrect data type. If this occurs during a LOAD, LOAD BIN, or LOAD ALL operation, then either the operation was not reading the file correctly, or the STORE, STORE BIN, or STORE ALL which originally wrote the file did not do so correctly, or the file was destroyed by an external disruption (such as a magnetic field). The error can also occur with an FREAD operation by trying to read into an array of a different data type than the stored array, or with trying to execute a GET on a non-DATA file.

ERROR 66 Excessive rejected tracks during a mass storage initialization. The medium could not be initialized for this reason and you will have to use another one. Medium wear, or a marginally-performing drive, is usually the reason for this happening.

ERROR 69 Format switch off. For disks with a "FORMAT" switch, it must be "on" for disk initialization to proceed.

ERROR 70 Not a disk interface. 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.

ERROR 71 Disk interface power is off. Turn it on. If it is already on, suspect that the interface is malfunctioning.

ERROR 72 Incorrect controller address, or the controller's power is off. If the former, change the address setting or change the program reference. If all is in order, suspect that the controller or its interface is malfunctioning. See the peripheral manual for location of the address switches.

ERROR 73 Incorrect device type in mass storage unit specifier. Check all your device settings and program references.

ERROR 74 Drive missing, or power off. Check your device settings and make sure that the device is receiving power.

ERROR 75 Disk system error. Possible power difficulties in interface or controller. Recycle power to those units. If necessary, reset the System 45.

ERROR 76 Incorrect unit code in mass storage unit specifier. Check all your device settings and program references.

ERROR 80 Cartridge out, or door open. Insert the cartridge or close the door. If that has already been done, the device is probably malfunctioning.

ERROR 81 Same as ERROR 75 above, except that it may apply to other devices than a hard disk.

ERROR 82 Mass storage device not present. Check your device settings and program references.

ERROR 83 Write-protected. The medium has been protected against recording. Check the write-protection devices on the medium or drive.

ERROR 84 Record not found. A bad spot on the medium has been encountered. You will either have to avoid this area of the medium by creating a "dummy" file to avoid attempts to use it or you will have to re-initialize the medium.

ERROR 85 Mass storage medium not initialized. Each medium must have been initialized through the INITIALIZE statement. If you are *certain* that the medium has been initialized and later hardware failures have not clobbered the initialization data, then you have a system failure. Otherwise, you must initialize the medium (remember, however, that initialization makes all previous data on the medium inaccessible).

ERROR 86 Not a System 45 data cartridge. Similar to ERROR 85 above.

ERROR 87 Record address error. This invariably is a hardware failure of the mass storage device. Clean the heads of the device; if this does not clear up the problem, perform the system test.

ERROR 88 Read data error. Usually the same causes as with ERROR 87.

ERROR 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.

Software Errors

Software-Related Errors

If you have ruled out a hardware problem as the source of an error, then the problem you have encountered is a software difficulty. As such, you should be able to handle the problem through your program's control. In order to give you some idea of what action(s) to take when a mass storage-related error develops, the following should be of some use –

ERROR 16 Dimensions are improper or inconsistent. 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.

ERROR 20 Integer overflow. 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 in the expression(s) used in the line with the error.

ERROR 38 TYP function not allowed. The TYP function is not allowed in output statements such as PRINT# or MAT PRINT#. Place the value in a variable and print the variable's value instead.

ERROR 39 Function subprogram not allowed. Such subprograms are not allowed in output statements such as PRINT# or MAT PRINT#. Place the value returned by the function into a variable and print the variable's value instead.

ERROR 46 No binary in STORE BIN, or no program in SAVE. There are no binaries currently in memory, or there are no program lines in memory between the limits you specified.

ERROR 50 File number out of range. The only permissible file numbers are 1 through 10, inclusive.

ERROR 51 File not currently assigned. You must execute an ASSIGN statement for that file number before using it.

ERROR 52 Improper mass storage unit specifier. Check particularly your syntax and the values of the select code.

ERROR 53 Improper file name. A file name may only be up to six characters long; the name must not contain the NULL, blank, quote-mark, colon, or DEL.

ERROR 54 Duplicate file name in directory. A file name may only appear once in a medium's directory. Choose another file name, or PURGE the old version. In some instances, RE-SAVE or RE-STORE may be an alternative statement to use.

ERROR 55 Directory overflow. Even though there may be space remaining on the medium, there is a maximum number of files the directory on a medium can handle (see the appendix for the particular device to find the number). Some files will have to be removed from the medium before any others can be added.

ERROR 56 File name is undefined. Couldn't find the file in the directory.

ERROR 58 Improper file type. Each statement works on only a certain type of file. Be certain the file you are trying to use is of the right type or that the statement you are trying to use is the right one.

ERROR 59 Physical or logical end-of-file. If you are in serial mode, this means you have run out of data. If you are in random mode, it means you are reading a record beyond the reserved file length or the specified **record number** is too large. This error can be trapped with the ON END statement.

ERROR 60 Physical or logical end-of-record in random access mode. Similar to ERROR 59 above. Attempting to read more data out of a single defined record in the random mode than is actually there.

ERROR 61 Defined-record size is too small for the data item. Either expand the defined-record size, or try to fit the data into more than one record.

ERROR 62 File is protected, or wrong protect code. You've got to use the proper protect code when trying to alter a protected file.

ERROR 63 The number of physical records is greater than 32 767. That's the limit; select something smaller.

ERROR 64 Medium overflow; out of storage space. There is not enough contiguous storage capacity on the medium to create the file. Use another medium, get rid of some of the files on the present one, or try to repack (as explained on pages 50-51).

ERROR 67 Mass storage parameter less than or equal to zero. You cannot have negative amounts of defined records or record sizes, or a negative record number. If you were using a variable as the parameter, check its value.

ERROR 68 Invalid line number in GET or LINK command. 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 the program in the file was created by another program and not by a SAVE or RE-SAVE command.

In addition to these numbered errors, the following error message can also appear during execution of READ# statements –

OVERFLOW IN LINE **nn**

This message appears only when you attempt to read a value into a short-precision or integer variable which is outside the range of the data type. The message implies "underflow" as well. This is not a fatal error; processing will continue and the value stored in the variable will be the default value for the data type. These values are –

Short-precision overflow	9.99999E63
Short-precision underflow	0
Integer overflow	32767
Integer underflow	0

Anticipating Software Errors

Software errors are ones which you, as the programmer, should be able to anticipate and control. The best way to handle such errors is to check things in advance before attempting an operation. For example, make sure data types are correct before attempting I/O operations. The TYP function can help you in that regard. And make sure parameters are within range if the parameters use variables. It may take an extra line or two to check such things, but they can avoid crippling errors during production runs.

If errors do slip past your checks in spite of your best efforts, then attempt to control their effects with the ON ERROR statement. Coupling this trapping facility with the ERRL and ERRN functions is an effective means of clearing up difficulties and allowing the program to retain control and decide whether to proceed or halt. The Operating and Programming Manual contains the information you need to utilize these statements.

When attempting to process after an error, there are two procedures recommended to ensure the integrity of your mass storage files. First, it is suggested that you get the file pointer to a known position. An error could possibly have misplaced the pointer in the file and you may not be reading or writing where you actually intend if you simply continue on. Instead, reposition the pointer to some known point, such as the beginning of the file (by reassigning it), or by executing a random READ# to some definite defined record.

Second, it is quite possible that your current buffered physical record (device buffer) may have been lost because of an error, particularly those which are hardware-caused. Thus, it is recommended that you try to re-generate the contents of the buffer when an error occurs.

Notes

Appendix A

ASCII Character Set

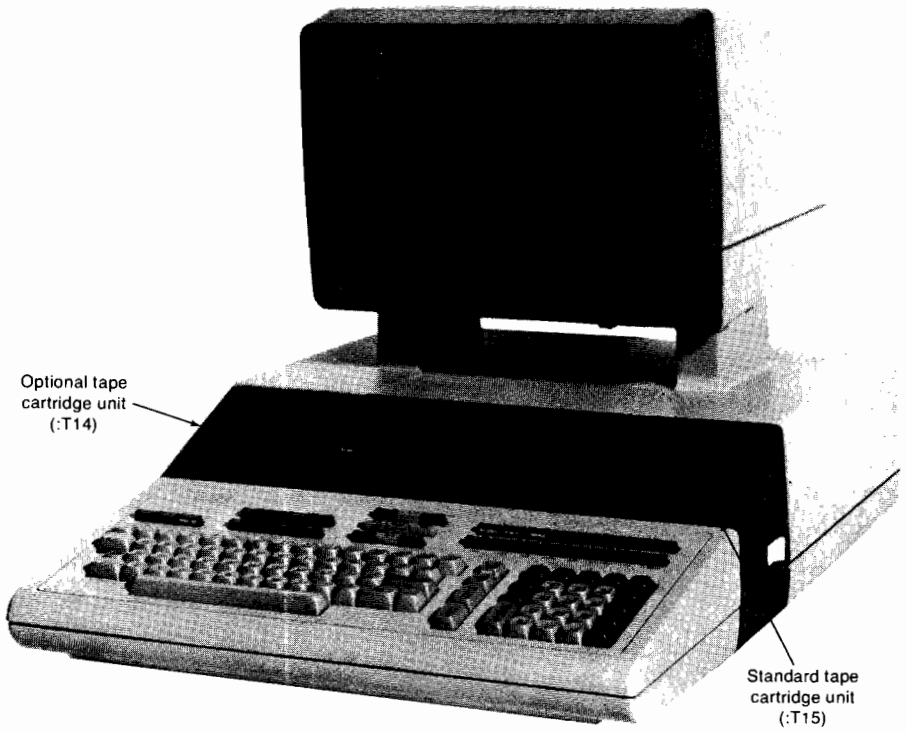
ASCII Char.	EQUIVALENT FORMS Binary	Octal	Dec	ASCII Char.	EQUIVALENT FORMS Binary	Octal	Dec	ASCII Char.	EQUIVALENT FORMS Binary	Octal	Dec	ASCII Char.	EQUIVALENT FORMS Binary	Octal	Dec
NULL	00000000	000	0	space	00100000	040	32	@	01000000	100	64	`	01100000	140	96
SOH	00000001	001	1	!	00100001	041	33	A	01000001	101	65	a	01100001	141	97
STX	00000010	002	2	"	00100010	042	34	B	01000010	102	66	b	01100010	142	98
ETX	00000011	003	3	#	00100011	043	35	C	01000011	103	67	c	01100011	143	99
EOT	00000100	004	4	$	00100100	044	36	D	01000100	104	68	d	01100100	144	100
ENQ	00000101	005	5	%	00100101	045	37	E	01000101	105	69	e	01100101	145	101
ACK	00000110	006	6	&	00100110	046	38	F	01000110	106	70	f	01100110	146	102
BELL	00000111	007	7	'	00100111	047	39	G	01000111	107	71	g	01100111	147	103
BS	00001000	010	8	(	00101000	050	40	H	01001000	110	72	h	01101000	150	104
HT	00001001	011	9	)	00101001	051	41	I	01001001	111	73	i	01101001	151	105
LF	00001010	012	10	*	00101010	052	42	J	01001010	112	74	j	01101010	152	106
V_{TAB}	00001011	013	11	+	00101011	053	43	K	01001011	113	75	k	01101011	153	107
FF	00001100	014	12	,	00101100	054	44	L	01001100	114	76	l	01101100	154	108
CR	00001101	015	13	–	00101101	055	45	M	01001101	115	77	m	01101101	155	109
SO	00001110	016	14	.	00101110	056	46	N	01001110	116	78	n	01101110	156	110
SI	00001111	017	15	/	00101111	057	47	O	01001111	117	79	o	01101111	157	111
DLE	00010000	020	16	ø	00110000	060	48	P	01010000	120	80	p	01110000	160	112
DC_1	00010001	021	17	1	00110001	061	49	Q	01010001	121	81	q	01110001	161	113
DC_2	00010010	022	18	2	00110010	062	50	R	01010010	122	82	r	01110010	162	114
DC_3	00010011	023	19	3	00110011	063	51	S	01010011	123	83	s	01110011	163	115
DC_4	00010100	024	20	4	00110100	064	52	T	01010100	124	84	t	01110100	164	116
NAK	00010101	025	21	5	00110101	065	53	U	01010101	125	85	u	01110101	165	117
SYNC	00010110	026	22	6	00110110	066	54	V	01010110	126	86	v	01110110	166	118
ETB	00010111	027	23	7	00110111	067	55	W	01010111	127	87	w	01110111	167	119
CAN	00011000	030	24	8	00111000	070	56	X	01011000	130	88	x	01111000	170	120
EM	00011001	031	25	9	00111001	071	57	Y	01011001	131	89	y	01111001	171	121
SUB	00011010	032	26	:	00111010	072	58	Z	01011010	132	90	z	01111010	172	122
ESC	00011011	033	27	;	00111011	073	59	[	01011011	133	91	{	01111011	173	123
FS	00011100	034	28	<	00111100	074	60	\	01011100	134	92	|	01111100	174	124
GS	00011101	035	29	=	00111101	075	61	]	01011101	135	93	}	01111101	175	125
RS	00011110	036	30	>	00111110	076	62	^	01011110	136	94	~	01111110	176	126
US	00011111	037	31	?	00111111	077	63	_	01011111	137	95	DEL	01111111	177	127

Appendix B

Internal Tape Cartridge

Two tape cartridge units are available internally with the System 45.

The standard unit has an **msus** of :T15 and is physically located above the keyboard on the right. An optional unit may be present; if so, it has an **msus** of :T14 and is physically located above the keyboard on the left.



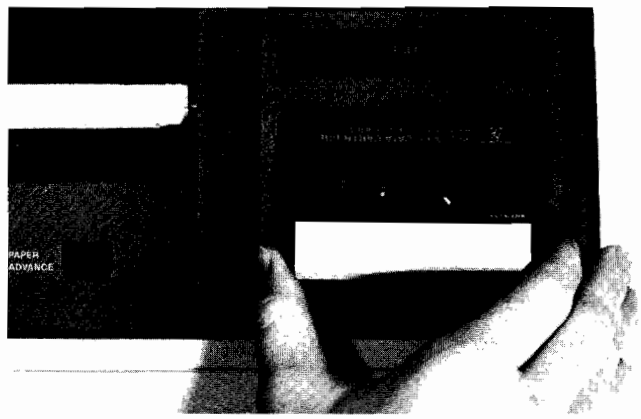
General Physical Information

Access rate (search speed)	14 300 bytes per second
Transfer rate (read/write)	1 800 bytes per second
Maximum rewind time	19 seconds
Initialization time	3 minutes
Tape length	42.67 metres (140 feet)
Number of tracks	2
Storage capacity	847 physical records
	216,832 bytes
	42 files
Typical tape life	50-100 hours
Typical error rate	1 in 10 000 000 bytes[1]

The Tape Cartridge

Inserting the Tape Cartridge

The tape cartridge should be inserted such that the label is up and facing outwards. Pressing the cartridge on the lower part of the unit-door (opaque part – see photograph below), will cause the door to open and accept the cartridge. Press in on the cartridge until the latch engages (there is a click). If the latch will not engage, press on the (eject) bar above the unit. The cartridge should then be able to be inserted and latched.



[1] Depending upon tape care and environment.

Removing the Tape Cartridge

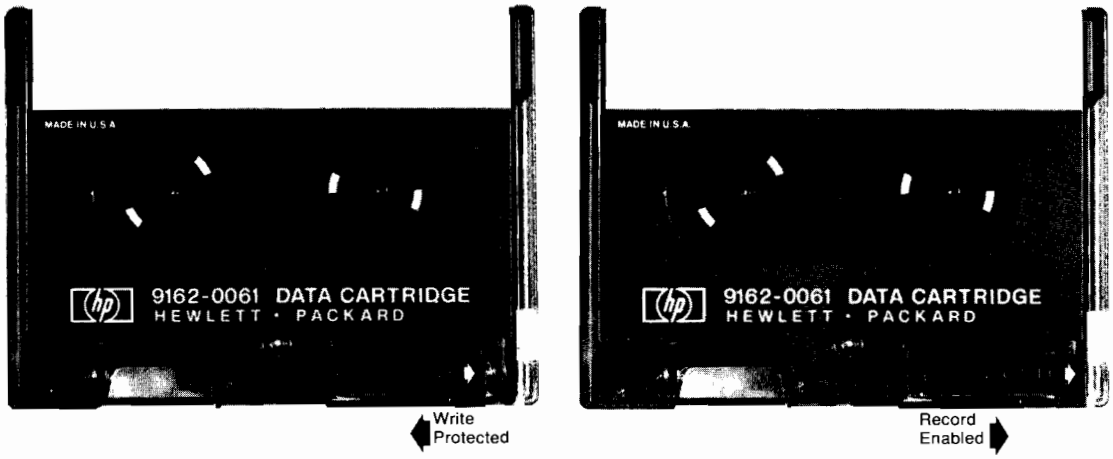
The cartridge may be removed by pressing on the bar above the unit. The unit will partially eject the cartridge and it may be freely removed the rest of the way.

CAUTION

DO NOT ATTEMPT TO REMOVE THE CARTRIDGE WHILE TAPE IS IN MOTION. DAMAGE TO THE TAPE MAY RESULT.

Write Protection

You may protect your cartridge against write (PRINT#) operations by sliding the Record Tab to the left before inserting the cartridge. To record on the cartridge, the tab must be all the way to the right.



Rewinding the Tape

The tape cartridge may be rewound to its starting point by using the REWIND statement. The form of the statement is –

REWIND [msus]

Operation of the desktop computer can take place while the tape is rewinding, provided it does not require use of the tape unit being rewound.

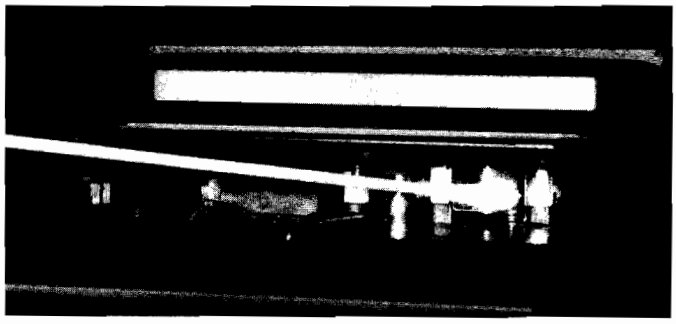
As a power-on definition, two special function keys have been allocated to REWIND commands. If the optional tape unit has been installed, key [k6] has been assigned REWIND":T14". Key [k7] has been assigned REWIND":T15". Pressing either of those keys will rewind the appropriate cartridge.

To stop a cartridge while it is rewinding, press reset ([CONTROL] [STOP]). But be careful in using this method – reset will halt operations on **every** device connected to the system, not just the cartridge concerned!

Tape Care

A cartridge which has become dirty through use or neglect is the greatest cause of cartridge-related errors. Several simple precautions can be taken to reduce the frequency of such difficulties –

- Clean the tape head and capstan (drive wheel) of the drive(s) after at least every eight hours of use – more frequently in particularly dirty environments. To clean, use a cotton applicator with head cleaning solution (HP part number 8500-1251) to wipe the tape head a few times. Be sure the head has dried before inserting the cartridge in the drive. (See photograph below.)



- Rewind the cartridge after each use.
- Keep the cartridge in the plastic container supplied with it.
- NEVER eject the tape cartridge while the tape is moving.

Other factors can effect the reliability of the cartridge. Strong magnetic fields can erase programs and data stored on the tape. Keeping it in a metal box, particularly during transport, should protect it. A simple card index box can serve for this purpose. Physical damage to the tape, such as a wrinkle or fold, can cause recording and loading problems. To reduce the probability of this happening, either remove the cartridge, or verify that the tape has been rewound, before turning on the computer.

Environmental Considerations

The HP-supplied cartridge has been tested for reliability between 0° and 45° C (32° to 113° F), and between 20% and 80% humidities (30° C maximum wet bulb temperature). Do not store the cartridge in environments where the temperatures or humidities may go outside these limits.

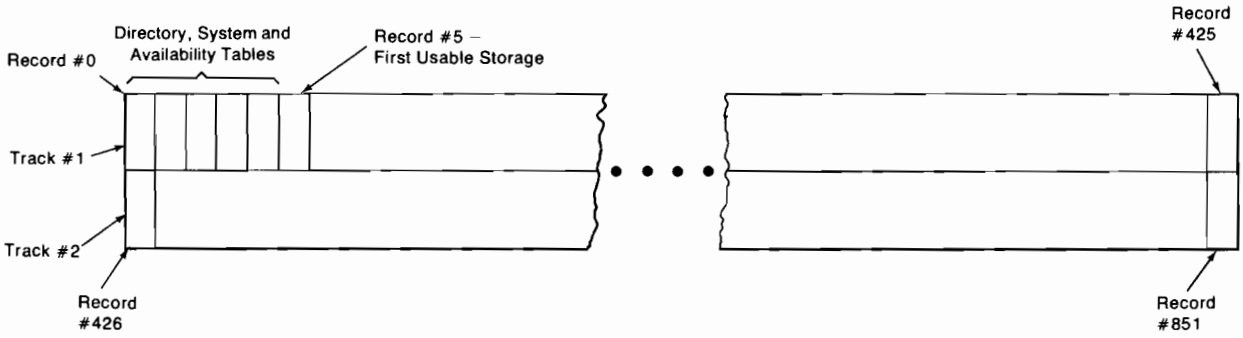
NOTE

Occasionally during the use of a tape cartridge, unexpected high-speed tape movement may occur. This action is to assure proper tape tension and does not affect processing or progams.

Tape Structure

The tape is organized with 2 tracks of 426 records each. The first five records are reserved for the directory, system, and availability tables.

The records are numbered consecutively from track to track. Thus record #426 is actually the first record on track 2. Diagramatically, the tape appears this way –



It is recommended that when recording files on the tape you avoid overlapping the file on both tracks, such as starting a four-record file with record #424. To avoid this happening inadvertently, it is recommended that you create a file one record long at record #425. This could be done right after initialization with a process something like this –

```
100  INITIALIZE ":T"
110  MASS STORAGE IS ":T"
120  CREATE "DUMMY",420
130  CREATE "++++++",1
140  PURGE "DUMMY"
150  END
```

Of course, it is possible to do something similar to this at any time after initialization and before using record 425. After this file is created, however, it is not possible to create files of more than 426 physical records until the file is purged.

Appendix C

Disk Drives

Hewlett-Packard manufactures a number of disk drives, in a wide variety of potential configurations, which may be used with the System 45. No program changes (except **msus**) or special considerations need be taken to transfer data from one device to another under the System 45's Unified Mass Storage Concept.

Interfacing

Physical connection between the System 45 and the hard disks discussed in this Appendix is accomplished with the 98041A Disc Interface. Installation, connection, and operation of the interface are discussed in the 98041A Disc Interface Installation Manual (HP part number 98041-90000). Once properly connected, the disk interface is invisible to you, and your commands and statements are made to the disk drives without concern for the interfacing unit.

Physical connection between the System 45 and the HP9885 Flexible Disk Drive is accomplished with the HP98032A Option 485 Interface. Installation and connection of the drive and interface are discussed in the HP 9885 Flexible Disk Drive Installation Manual (HP part number 09885-90010).

Available Devices

The following disk drives are available for use with the System 45: the HP7905A Disc Drive; the HP7906 Disc Drive; the HP7920 Disc Drive; and the HP9885 Flexible Disk Drive. Each drive is supplied with an operating manual to which reference can be made for any information regarding installation, operation, and maintenance of the drive. Refer to the manual provided with your drive for particular information on the characteristics of the drive which might affect your programming.

As a quick reference, the following table gives some of the general operating characteristics for the available drives which may have an affect upon your program planning or system design –

	7905A	7906	7920	9885
Type of disk	Hard	Hard	Hard	Flexible
Storage Capacity:				
Bytes	4 866 048 Fixed 9 732 096 Removable	9 732 096 Fixed 9 732 096 Removable	48 758 784	499 200
Physical records	19 008 Fixed 38 016 Removable	38 016 Fixed 38 016 Removable	190 464	1 950
Tracks	396 Fixed 792 Removable	792 Fixed 792 Removable	3 968	65
Files	1 136 Fixed 2 288 Removable	2 288 Fixed 2 288 Removable	8 000	352
Bytes per record	256	256	256	256
Records per track	48	48	48	30
Tracks per surface	400	800 Fixed 400 Removable	800	67
Accessing:				
Rate of spin	3 600 rpm	3 600 rpm	3 600 rpm	360 rpm
Access mode	Surface	Surface	Cylinder	--
Number of heads (surfaces)	1 Fixed 2 Removable	1 Fixed 2 Removable	5	1
Maximum transfer rate (bytes per second)	937 500	937 500	937 500	46 000
Average seek time	25 ms*	25 ms*	25 ms*	267 ms
Average rotational delay	8.3 ms	8.3 ms	8.3 ms	83 ms
Head settling time	--	--	--	8 ms
Step time	5 ms*	5 ms*	5 ms*	8 ms

*includes head settling time

Disk System Test

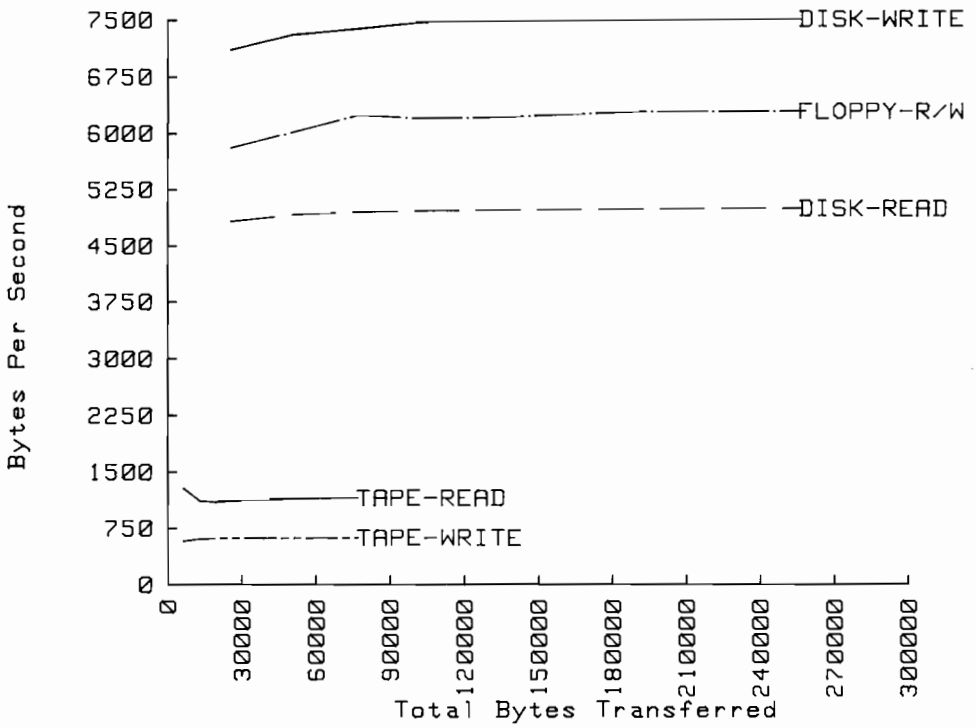
The system tests for all disks, hard and flexible, can be found in the system test booklet.

Timings

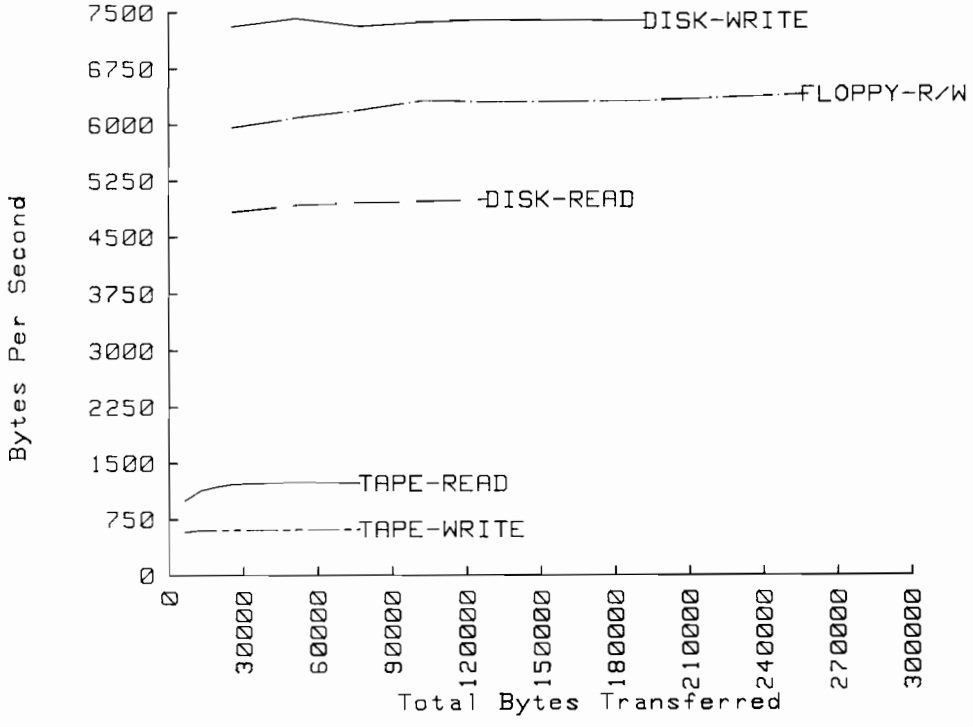
The following graphs present the transfer rates in various situations. It is included as an aid in designing those applications where efficiency or speed is a major consideration. All flexible disk timings utilize the default interleave factor (7).

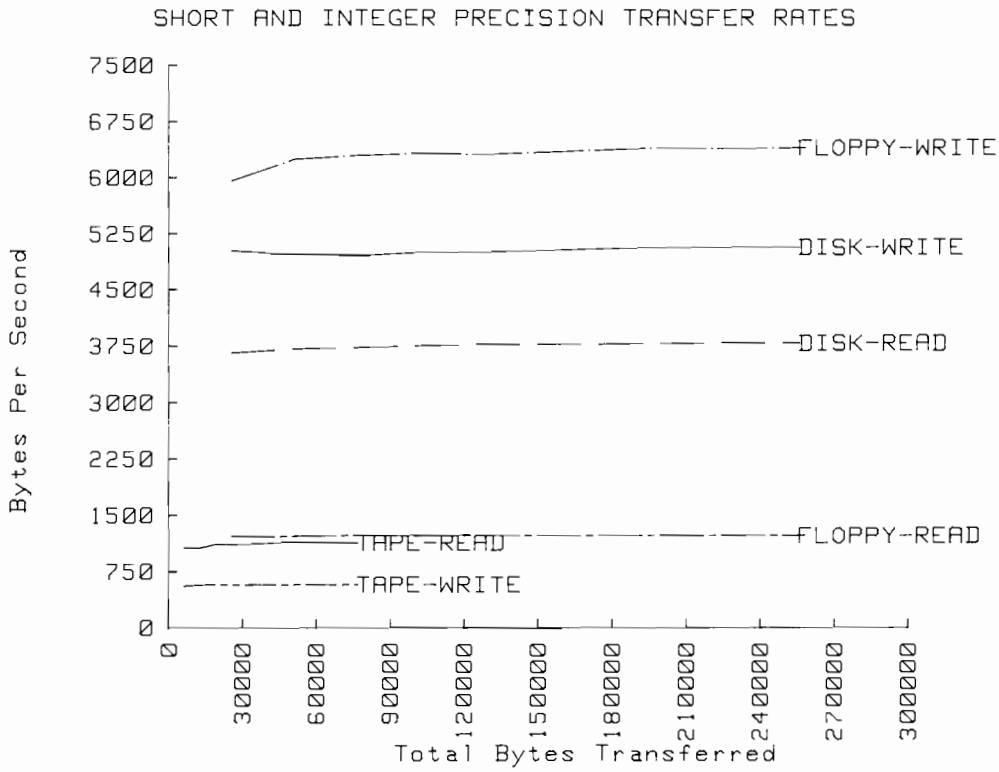
The graphs were produced under program control, utilizing the System 45 CRT/Graphics System.

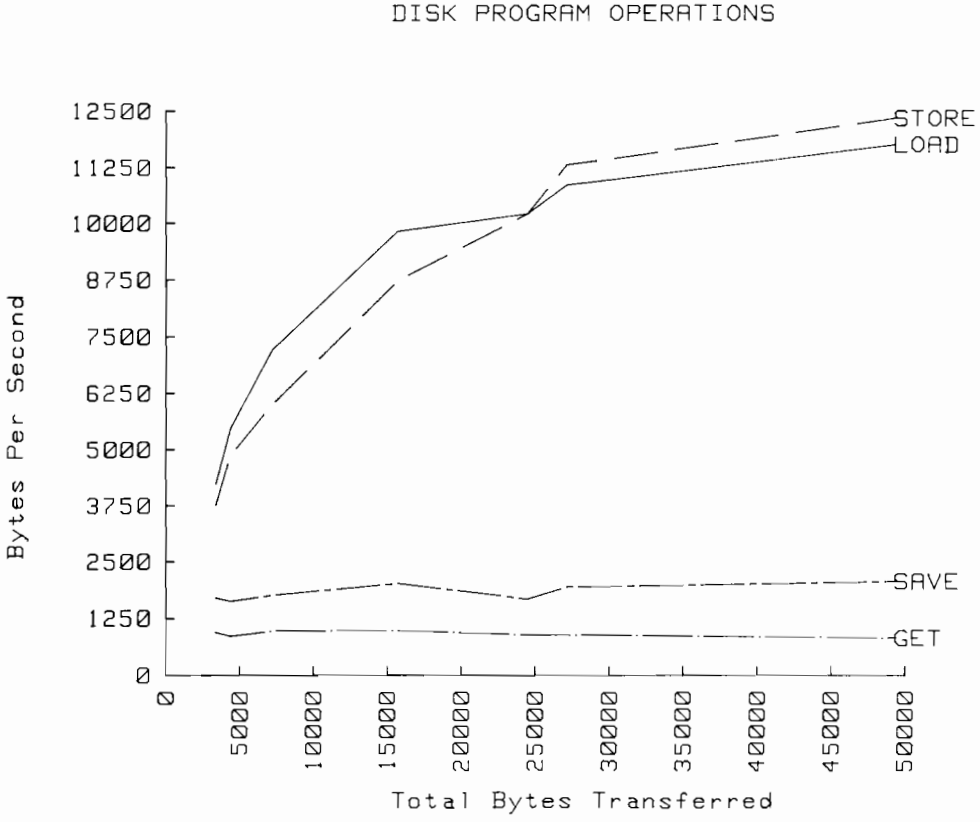
STRING TRANSFER RATES



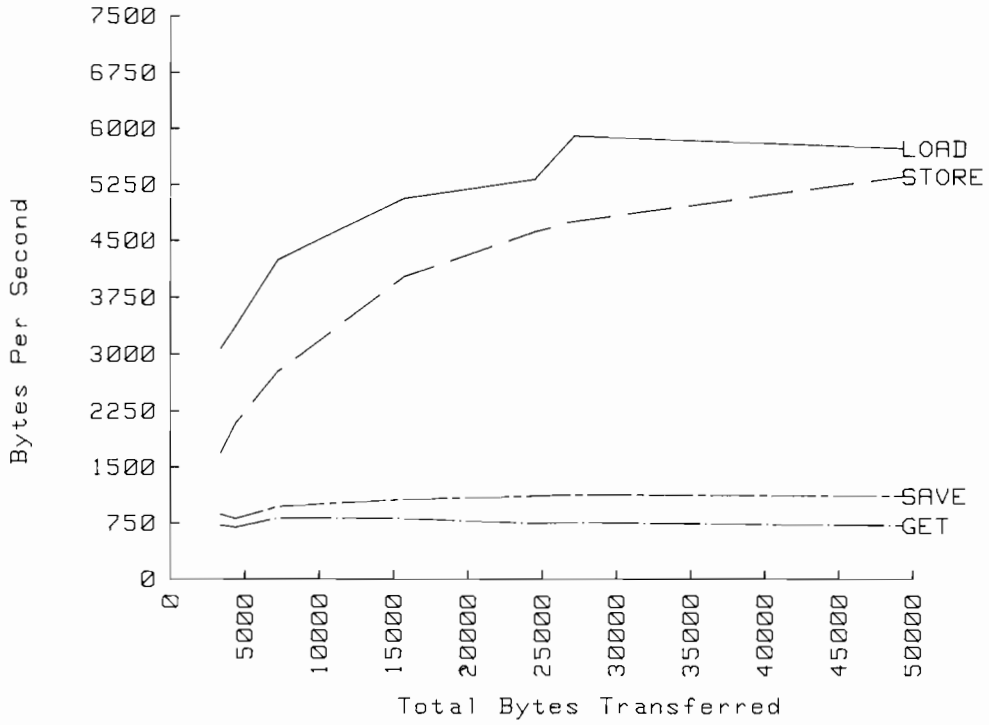
FULL-PRECISION TRANSFER RATES



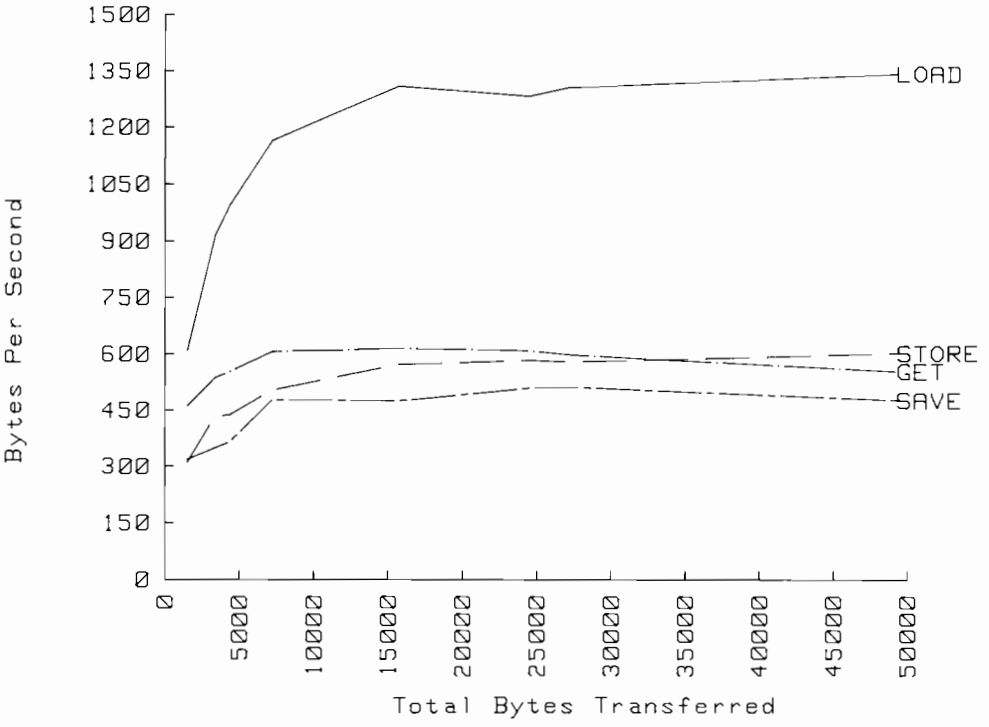

SHORT AND INTEGER PRECISION TRANSFER RATES
Bytes Per Second
7500
6750
6000
5250
4500
3750
3000
2250
1500
750
0
FLOPPY-WRITE
DISK-WRITE
DISK-READ
FLOPPY-READ
TAPE-READ
TAPE-WRITE
0
30000
60000
90000
120000
150000
180000
210000
240000
270000
300000
Total Bytes Transferred

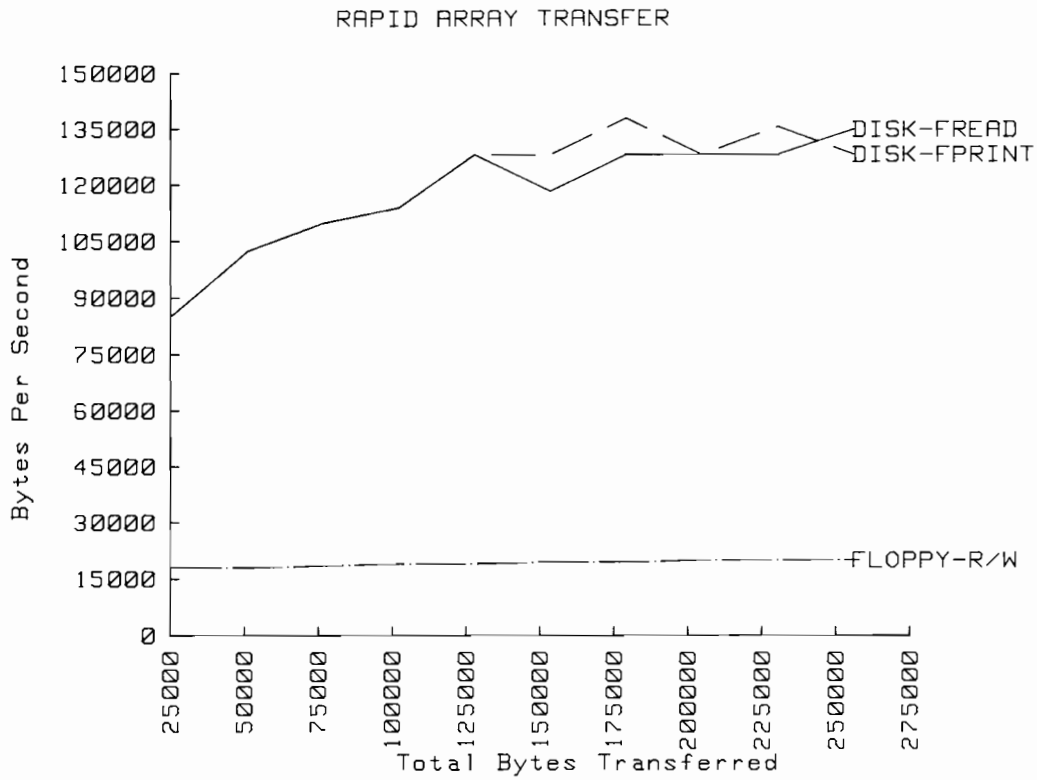

DISK PROGRAM OPERATIONS
Bytes Per Second
12500
11250
10000
8750
7500
6250
5000
3750
2500
1250
0
STORE
LOAD
SAVE
GET
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
Total Bytes Transferred

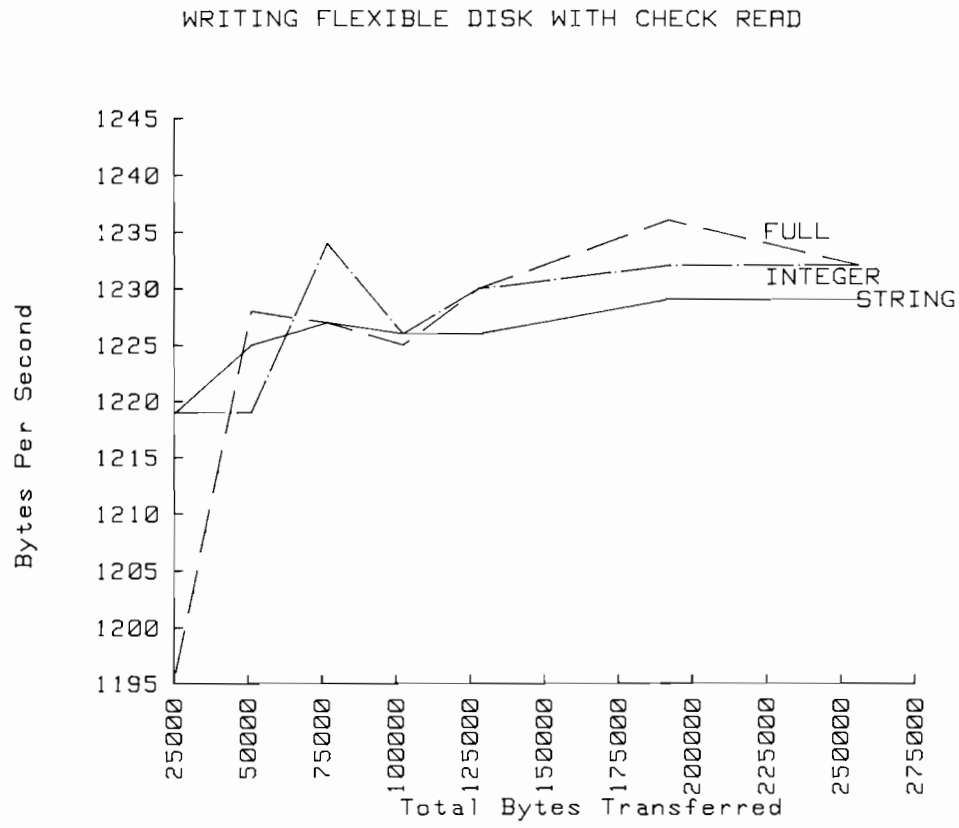
FLEXIBLE DISK PROGRAM OPERATIONS

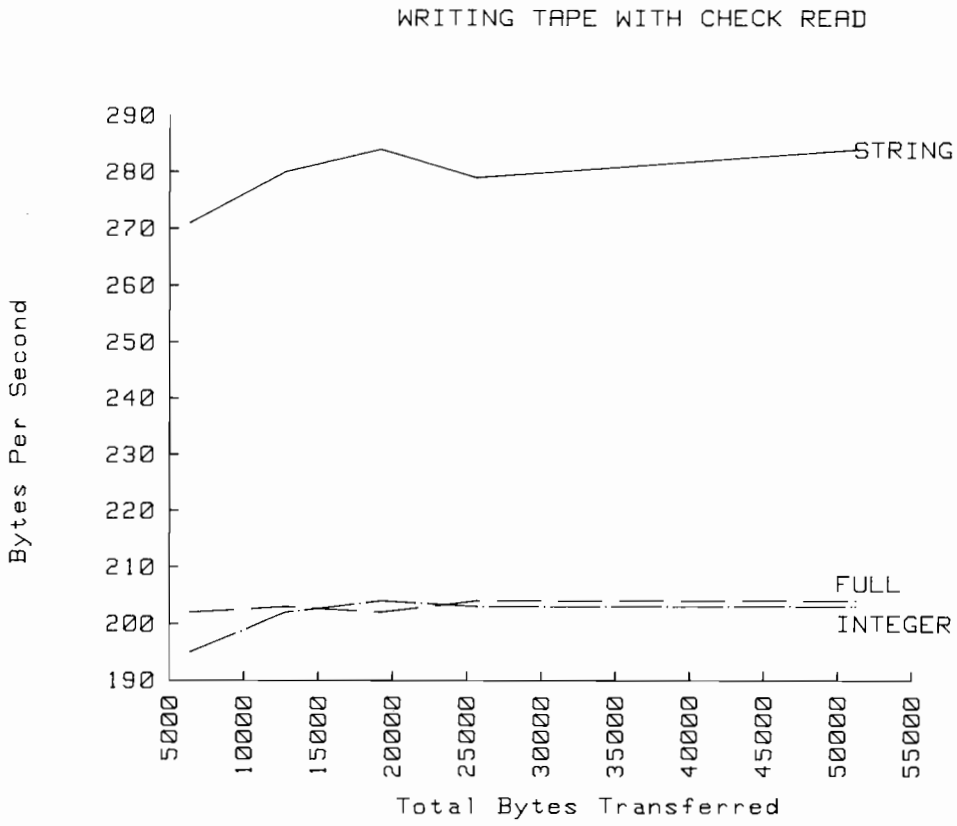


TAPE PROGRAM OPERATIONS




RAPID ARRAY TRANSFER
Bytes Per Second
150000
135000
120000
105000
90000
75000
60000
45000
30000
15000
0
DISK-FREAD
DISK-FPRINT
FLOPPY-R/W
25000
50000
75000
100000
125000
150000
175000
200000
225000
250000
275000
Total Bytes Transferred


WRITING FLEXIBLE DISK WITH CHECK READ
Bytes Per Second
1245
1240
1235
1230
1225
1220
1215
1210
1205
1200
1195
FULL
INTEGER
STRING
25000
50000
75000
100000
125000
150000
175000
200000
225000
250000
275000
Total Bytes Transferred


WRITING TAPE WITH CHECK READ
Bytes Per Second
290
280
270
260
250
240
230
220
210
200
190
STRING
FULL
INTEGER
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
55000
Total Bytes Transferred

Appendix D

Mass Storage Command Basic Syntax

The following is an alphabetical list of the mass storage statements and their basic syntaxes. For a full discussion of their semantical meanings and applications, consult the indicated pages in this manual or in the Operating and Programming Manual.

Mass Storage Unit Specifier (msus) (pages 5-7)

: device code [select code [, controller address | 9885 unit code [, unit code]]]

where **device code** is a capital letter designating the type of peripheral. Permissible codes are –

Code	Device
C	Removable disk cartridge (HP7906)
D	Fixed disk (HP7906)
F	Flexible disk (HP9885)
P	Disk pack (HP7920)
T	Tape cartridge (internal to the computer)
Y	Removable disk cartridge (HP7905A)
Z	Hard disk (HP7905A)

select code is an integer in the range of 1 through 12 (plus 14 and 15 for tape cartridges only).

controller address is an integer in the range of 0 to 7.

9885 unit code is an integer in the range of 0 through 3.

unit code is an integer in the range of 0 through 7.

File Specifier (page 10)

file name [msus]

where **file name** is a 1- to 6-character string expression containing any character except –

NULL	ASCII decimal value 0
Quote-mark (")	ASCII decimal value 34
Colon (:)	ASCII decimal value 58
(unnamed)	ASCII decimal value 255

Line Identifier (see Operating and Programming Manual)

a line identifier is a positive *integer* (not a numeric expression), less than 10000, which represents a line number, or it is a *name* which represents a *label*.

ASSIGN (pages 31-33)

ASSIGN # file number TO file specifier [, return variable [, protect code]]
ASSIGN file specifier TO # file number [, return variable [, protect code]]
ASSIGN # file number TO * [, return variable[, protect code]]
ASSIGN * TO # file number [, return variable [, protect code]]

where **file number** is a numeric expression; **return variable** is a non-string variable; and **protect code** is a string expression.

BUFFER (pages 59-61)

BUFFER # file number

where **file number** is a numeric expression.

CAT (pages 21-23)

CAT [selective catalog specifier / msus [, heading suppression]]
CAT # select code [, HP-IB device address][; selective catalog specifier / msus [, heading suppression]]

where **selective catalog specifier** and **msus** (either separately or in combination) form a single string expression; and **select code**, **HP-IB device address**, and **heading suppression** are numeric expressions.

CHECK READ (pages 34-36)

CHECK READ [OFF] [# file number]

where **file number** is a numeric expression.

COPY (page 50)

COPY source-file specifier TO destination-file specifier [, protect code]

where **protect code** is a string expression.

CREATE (pages 29-30)

CREATE file specifier , number of defined records [, record length]

where **number of defined records** and **record length** are numeric expressions.

FCREATE (pages 39-40)

FCREATE file specifier , number of physical records

where **number of physical records** is a numeric expression.

FPRINT (pages 40-41)

FPRINT file specifier [, protect code] , array identifier

where **protect code** is a string expression; and **array identifier** is a name followed by (*).

FREAD (pages 40-41)

FREAD file specifier [, protect code] , array identifier

where **protect code** is a string expression; and **array identifier** is a name followed by (*).

GET (pages 71-74)

GET file specifier [, line identifier [, execution-line identifier]]

INITIALIZE (pages 13-14)

INITIALIZE msus [, interleave factor]

where **interleave factor** is a numeric expression.

LINK (pages 72-73)

LINK file specifier [, line identifier [, execution-line identifier]]

LOAD (page 70)

LOAD file specifier [, execution-line identifier]

LOAD ALL (pages 75-76)

LOAD ALL file specifier

LOAD BIN (page 75)

LOAD BIN file specifier

LOAD KEY (pages 74-75)

LOAD KEY file specifier

MASS STORAGE IS (page 8)

MASS STORAGE IS msus

OFF END (page 49)

OFF END# file number

where **file number** is a numeric expression.

ON END (page 49)

ON END# file number CALL subprogram name
ON END# file number GOSUB line identifier
ON END# file number GOTO line identifier

where **file number** is a numeric expression; **subprogram name** is a name.

OVERLAP (pages 66-67)

OVERLAP

PRINT# (pages 33-39)

[MAT] PRINT# file number [, defined-record number] [; data list] [[,]END]

where **file number** and **defined-record number** are numeric expressions.

PROTECT (page 52)

PROTECT file specifier , protect code

where **protect code** is a string expression.

PURGE (pages 50-51)

PURGE file specifier [, protect code]

where **protect code** is a string expression.

READ# (pages 36-39)

[MAT] READ# file number [, defined-record number] [; variable list]

where **file number** and **defined-record number** are numeric expressions.

RENAME (page 53)

RENAME old-file specifier TO new file name [, protect code]

where **old-file specifier**, **new file name** and **protect code** are string expressions.

RE-SAVE (page 71)

RE-SAVE file specifier [, protect code] [, first-line identifier [, final-line identifier]]

where **protect code** is a string expression.

RE-STORE (page 69)

RE-STORE file specifier [, protect code]

where **protect code** is a string expression.

REWIND (page 92)

REWIND [msus]

SAVE (page 70)

SAVE file specifier [, first-line identifier [, final-line identifier]]

SERIAL (pages 66-67)

SERIAL

STORE (page 69)

STORE file specifier

STORE ALL (pages 75-76)

STORE ALL file specifier

STORE BIN (page 75)

STORE BIN file specifier

STORE KEY (page 74)

STORE KEY file specifier

TYP Function (pages 42-43)

TYP (file number)

where **file number** is a numeric expression.

Appendix E

Error Messages

The following is a numerical list of the system error messages. A brief description of the error is given. For those errors involving the mass storage statements and system, also consult Chapter 6. For all other errors, reference the Operating and Programming Manual.

1	Missing ROM or configuration error
2	Memory overflow
3	Line not found or not in current program segment
4	Improper return
5	Abnormal program termination
6	Improper FOR/NEXT matching
7	Undefined function or subroutine
8	Improper parameter matching
9	Improper number of parameters
10	String value required
11	Numeric value required
12	Attempt to redeclare variable
13	Array dimensions not specified
14	Multiple OPTION BASE statements or OPTION BASE statement preceded by variable declarative statements
15	Invalid bounds on array dimension or string length in memory allocation statement
16	Dimensions are improper or inconsistent
17	Subscript out of range
18	Substring out of range or string too long
19	Improper value

20	Integer precision overflow
21	Short precision overflow
22	Real precision overflow
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
32	String does not represent valid number or string response when numeric data required
33	Improper argument for NUM, CHR$, or RPT$ function
34	Referenced line is not IMAGE statement
35	Improper format string
36	Out of DATA
37	EDIT string longer than 160 characters
38	I/O function not allowed
39	Function subprogram not allowed
40	Improper replace, delete or REN command
41	First line number greater than second
42	Attempt to replace or delete a busy line or subprogram
43	Matrix not square
44	Illegal operand in matrix transpose or matrix multiply
45	Nested keyboard entry statements

46	No binary in STORE BIN or no program in SAVE
47	Subprogram COM declaration is not consistent with main program
48	Recursion in single line function
49	Line specified in ON declaration not found
50	File number less than 1 or greater than 10
51	File not currently assigned
52	Improper mass storage unit specifier
53	Improper file name
54	Duplicate file name
55	Directory overflow
56	File name is undefined
57	Mass Storage ROM is missing
58	Improper file type
59	Physical or logical end-of-file found
60	Physical or logical end-of-record found in random mode
61	Defined record size is too small for data item
62	File is protected or wrong protect code specified
63	The number of physical records is greater than 32767
64	Medium overflow (out of user storage space)
65	Incorrect data type
66	Excessive rejected tracks during a mass storage initialization
67	Mass storage parameter less than or equal to 0
68	Invalid line number in GET or LINK operation
69 - 79	See Mass Storage ROM errors
80	Cartridge out or door open
81	Mass storage device failure
82	Mass storage device not present

83	Write protected
84	Record not found
85	Mass storage medium is not initialized
86	Not a compatible tape data cartridge
87	Record address error
88	Read data error
89	Check read error
90	Mass storage system error
91-99	See Mass Storage ROM errors
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
104-109	Unused
110-119	See Graphics ROM errors

SYSTEM ERROR
System Error octal number

These two errors indicate an error in the machine's firmware system; they are fatal errors. If reset does not bring control back, the machine must be turned off, then on again. If the problem persists, contact your 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 is an example. The first message is –

I/O ERROR ON SELECT CODE select code

If the condition is not correct, the machine beeps intermittently and the following message replaces the first –

DEVICE TIMEOUT ON SELECT CODE select code

The I/O device can be made usable by correcting the error (loading paper for example), then executing the READY# command –

READY# select code

This command readies the I/O device and the operation which was attempted is completed.

Mass Storage ROM Errors

69	Format switch off
70	Not a disc interface
71	Disc interface power off
72	Incorrect controller address, or controller power off
73	Incorrect device type in mass storage unit specifer
74	Drive missing or power off
75	Disc system error
76	Incorrect unit code in mass storage unit specifier
77-79	Unused
91-99	Unused

Graphics ROM Errors

110	Plotter specifications not recognized.
111	Plotter not previously specified.
112	CRT Graphics hardware not installed.
113	LIMIT specifications out of range.
114-119	Unused

Appendix F

Maintenance

Maintenance Agreements

Service is an important factor when you buy Hewlett-Packard equipment. If you are to get maximum use from your equipment, it must be in good working order. An HP Maintenance Agreement is the best way to keep your equipment in optimum running condition.

Consider these important advantages –

- Fixed Cost – The cost is the same regardless of the number of calls, so it is a figure that you can budget.
- Priority Service – Your Maintenance Agreement assures that you receive priority treatment, within an agreed-upon response time.
- On-Site Service – There is no need to package your equipment and return it to HP. Fast and efficient modular replacement at your location saves you both time and money.
- A Complete Package – A single charge covers labor, parts, and transportation.
- Regular Maintenance – Periodic vists are included, per factory recommendations, to keep your equipment in optimum operating condition.
- Individualized Agreements – Each Maintenance Agreement is tailored to support your equipment configuration and your requirements.

After considering these advantages, we are sure you will see that a Maintenance Agreement is an important and cost-effective investment.

For more information, please contact your local HP Sales and Service Office. A list follows.

HEWLETT hp PACKARD

SALES & SERVICE OFFICES

AFRICA, ASIA, AUSTRALIA

ANGOLA
Telectra
Empresa Técnica de Equipamentos Eléctricos, S.A.R.L.
R. Barbosa Rodrigues, 42-I OT
Caixa Postal, 6487
Luanda
Tel. 35515/6
Cable: TELECTRA Luanda

AUSTRALIA
Hewlett-Packard Australia Pty. Ltd.
31-41 Joseph Street
Blackburn, Victoria 3130
P.O. Box 36
Doncaster East, Victoria 3109
Tel: 89-6351
Telex: 31-024
Cable: HEWPARD Melbourne

Hewlett-Packard Australia Pty. Ltd.
31 Bridge Street
Pymble
New South Wales, 2073
Tel: 449-6566
Telex: 21561
Cable: HEWPARD Sydney

Hewlett-Packard Australia Pty. Ltd.
153 Greenhill Road
Parkside, S.A., 5063
Tel: 272-5911
Telex: 82536 ADEL
Cable: HEWPARD ADELAID

Hewlett-Packard Australia Pty. Ltd.
141 Stirling Highway
Nedlands, W.A. 6009
Tel: 86-5455
Telex: 93859 PERTH
Cable: HEWPARD PERTH

Hewlett-Packard Australia Pty. Ltd.
121 Wollongong Street
Fyshwick, A.C.T. 2609
Tel: 95-2733
Telex: 62650 Canberra
Cable: HEWPARD CANBERRA

Hewlett Packard Australia Pty. Ltd.
5th Floor
Teachers Union Building
495-499 Boundary Street
Spring Hill, 4000 Queensland
Tel: 229-1544
Cable: HEWPARD Brisbane

GUAM
Medical/Pocket Calculators Only
Guam Medical Supply, Inc.
Jay Ease Building, Room 210
P.O. Box 8947
Tamuning 96911
Tel: 646-4513
Cable: EARMEO Guam

HONG KONG
Schmidt & Co. (Hong Kong) Ltd.
P.O. Box 297
Connaught Centre
39th Floor
Connaught Road, Central
Hong Kong
Tel: H-255291-5
Telex: 74766 SCHMC HX
Cable: SCHMIDTCO Hong Kong

INDIA
Blue Star Ltd.
Kasturi Buildings
Jamshedji Tata Rd.
Bombay 400 020
Tel: 29 50 21
Telex: 001-2156
Cable: BLUEFROST

Blue Star Ltd.
Sahas
414/2 Vir Savarkar Marg
Prabhadevi
Bombay 400 025
Tel: 45 78 87
Telex: 011-4093
Cable: FROSTBLUE

Blue Star Ltd.
Band Box House
Prabhadevi
Bombay 400 025
Tel: 45 73 01
Telex: 011-3751
Cable: BLUESTAR

Blue Star Ltd.
7 Hare Street
P.O. Box 506
Calcutta 700 001
Tel: 23-0131
Telex: 021-7655
Cable: BLUESTAR

Blue Star Ltd.
7th & 8th Floor
Bhandari House
91 Nehru Place
New Delhi 110024
Tel: 634770 & 635166
Telex: 031-2463
Cable: BLUESTAR

Blue Star Ltd.
Blue Star House
11/11A Magarath Road
Bangalore 560 025
Tel: 55868
Telex: 043-430
Cable: BLUESTAR

Blue Star Ltd.
Meeakshi Mandiran
xxx/1678 Mahatma Gandhi Rd.
Cochin 682 016
Tel: 32069,32161,32282
Telex: 0685-514
Cable: BLUESTAR

Blue Star Ltd.
1-1-117/1
Sarojini Devi Road
Secunderabad 500 003
Tel: 70126, 70127
Cable: BLUEFROST
Telex: 015-459

Blue Star Ltd.
2/34 Kodambakkam High Road
Madras 600034
Tel: 82056
Telex: 041-379
Cable: BLUESTAR

INDONESIA
BERCA Indonesia P.T.
P.O. Box 496/Jkt.
JLN=Abdul Muis 62
Jakarta
Tel: 40369, 49886,49255,356038
JKT 42895
Cable: BERCACON

BERCA Indonesia P.t.
63 JL. Raya Gubeng
Surabaya
Tel: 44309

ISRAEL
Electronics & Engineering Div. of Motorola Israel Ltd.
17, Kremenetski Street
P.O. Box 25016
Tel-Aviv
Tel: 38973
Telex: 33569
Cable: BASTEL Tel-Aviv

JAPAN
Yokogawa-Hewlett-Packard Ltd.
Ohashi Building
59-1 Yoyogi 1-Chome
Shibuya-ku, **Tokyo** 151
Tel: 03-370-2281/92
Telex: 232-2024YHP MARKET TOK 23-724
Cable: YHPMARKET

Yokogawa-Hewlett-Packard Ltd.
Chuo Bldg., 4th Floor
4-20, Nishinakajima 5-chome
Yodogawa-ku, Osaka-shi
Osaka 532
Tel: 06-304-6021

Yokogawa-Hewlett-Packard Ltd.
Nakamo Building
24 Kami Sasajima-cho
Nakamura-ku, **Nagoya**, 450
Tel: (052) 571-5171

Yokogawa-Hewlett-Packard Ltd.
Tanigawa Building
2-24-1 Tsuruya-cho
Kanagawa-ku
Yokohama, 221
Tel: 045-312-1252
Telex: 382-3204 YHP YOK

Yokogawa-Hewlett-Packard Ltd.
Mito Mitsu Building
105, Chome-1,San-no-maru
Mito, Ibaragi 310
Tel: 0292-25-7470

Yokogawa-Hewlett-Packard Ltd.
Inoue Building
1348-3, Asahi-cho, 1-chome
Atsugi, Kanagawa 243
Tel: 0462-24-0452

Yokogawa-Hewlett-Packard Ltd.
Kumagaya Asahi
Hackiuni Building
4th Floor
3-4, Tsukuba
Kumagaya, Saitama 360
Tel: 0485-24-6563

KENYA
Technical Engineering Services(E.A.)Ltd.,
P.O. Box 18311
Nairobi
Tel: 557726/556762
Cable: PROTON

Medical Only
International Aeradio(E.A.)Ltd.,
P.O. Box 19012
Nairobi Airport
Nairobi
Tel: 336055/56
Telex: 22201/22301
Cable: INTAERIO Nairobi

KOREA
Samsung Electronics Co., Ltd.
20th Fl. Dongbang Bldg. 250, 2-KA
C.P.O. Box 2775
Taepyung-Ro, Chung-Ku
Seoul
Tel: (23) 6811
Telex: 22575
Cable: ELEKSTAR Seoul

MALAYSIA
Teknik Mutu Sdn. Bhd.
2 Lorong 13/6A
Section 13
Petaling Jaya, **Selangor**
Tel: 54994/54916
Telex: MA 37605

Protel Engineering
P.O. Box 1917
Lot 259, Satok Road
Kuching, **Sarawak**
Tel: 2400
Cable: PROTEL ENG

MOZAMBIQUE
A.N. Goncalves, Lta.
162, 1 Apt. 14 Av. O. Luis
Caixa Postal 107
Lourenco Marques
Tel: 27091, 27114
Telex: 6-203 NEGON Mo
Cable: NEGON

NEW ZEALAND
Hewlett-Packard (N.Z.) Ltd.
P.O. Box 9443
Courtenay Place
Wellington
Tel: 877-199
Cable: HEWPACK Wellington

Hewlett-Packard (N.Z.) Ltd.
Pakuranga Professional Centre
267 Pakuranga Highway
Box 51092
Pakuranga
Tel: 569-651
Cable: HEWPACK,Auckland

Analytical/Medical Only
Medical Supplies N.Z. Ltd.
Scientific Division
79 Carlton Gore Rd., Newmarket
P.O. Box 1234
Auckland
Tel: 75-289
Cable:DENTAL Auckland

Analytical/Medical Only
Medical Supplies N.Z. Ltd.
P.O. Box 1994
147-161 Tory St.
Wellington
Tel: 850-799
Telex: 3858
Cable: DENTAL, Wellington

Analytical/Medical Only
Medical Supplies N.Z. Ltd.
P.O. Box 309
239 Stanmore Road
Christchurch
Tel: 892-019
Cable: DENTAL, Christchurch

Analytical/Medical Only
Medical Supplies N.Z. Ltd.
303 Great King Street
P.O. Box 233
Dunedin
Tel: 88-817
Cable: DENTAL, Dunedin

NIGERIA
The Electronics Instrumentations Ltd.
N6B/770 Oyo Road
Oluseun House
P.M.B. 5402
Ibadan
Tel: 61577
Telex: 31231 TEIL Nigeria
Cable: THETEIL Ibadan

The Electronics Instrumentations Ltd.
144 Agege Motor Road, Mushin
P.O. Box 6645
Lagos
Cable: THETEIL Lagos

PAKISTAN
Mushko & Company, Ltd.
Oosman Chambers
Abdullah Haroon Road
Karachi-3
Tel: 511027, 512927
Telex: 2894
Cable: COOPERATOR Karachi

Mushko & Company, Ltd.
38B, Satellite Town
Rawalpindi
Tel: 41924
Cable: FEMUS Rawalpindi

PHILIPPINES
The Online Advanced Systems Corporation
Rico House
Amorsolo cor. Herrera Str.
Legaspi Village, Makati
Metro **Manila**
Tel: 85-35-81, 85-34-91
Telex: 3274 ONLINE

RHODESIA
Field Technical Sales
45 Kelvin Road North
P.O. Box 3458
Salisbury
Tel: 705231 (5 lines)
Telex: RH 4122

SINGAPORE
Hewlett-Packard Singapore (Pte.) Ltd.
1150 Depot Road
Alexandra P.O. Box 58
Singapore 4
Tel: 270-2355
Telex: HPSG RS 21486
Cable: HEWPACK, Singapore

SOUTH AFRICA
Hewlett-Packard South Africa (Pty.), Ltd.
Private Bag Wendywood
Sandton, Transvaal 2144
Hewlett-Packard Centre
Daphne Street, Wendywood,
Sandton, Transvaal 2144
Tel: 802-10408
Telex: 8-4782
Cable: HEWPACK JOHANNESBURG

Service Department
Hewlett-Packard South Africa (Pty.), Ltd.
P.O. Box 39325
Gramley, Sandton, 2018
451 Wynberg Extension 3,
Sandton, 2001
Tel: 636-8188/9
Telex: 8-2391

Hewlett-Packard South Africa (Pty.), Ltd.
P.O. Box 120
Howard Place, Cape Province, 7450
Pine Park Centre, Forest Drive,
Pinelands, Cape Province, 7405
Tel: 53-7955 thu 9
Telex: 57-0006

Service Department
Hewlett-Packard South Africa (Pty.), Ltd.
P.O. Box 37099
Overport, Durban 4067
Braby House
641 Ridge Road
Durban, 4001
Tel: 88-7478
Telex: 6-7954

TAIWAN
Hewlett-Packard Far East Ltd.,
Taiwan Branch
39 Chung Hsiao West Road
Sec. 1, 7th Floor
Taipei
Tel: 3819160-4
Cable: HEWPACK TAIPEI

Hewlett-Packard Far East Ltd.
Taiwan Branch
68-2, Chung Cheng 3rd. Road
Kaohsiung
Tel: (07) 242318-Kaohsiung

Analytical Only
San Kwang Instruments Co., Ltd.,
No. 20, Yung Sui Road
Taipei
Tel: 3715171-4 (5 lines)
Telex: 22894 SANKWANG
Cable: SANKWANG TAIPEI

TANZANIA
Medical Only
International Aeradio (E.A.), Ltd.
P.O. Box 861
Dar es Salaam
Tel: 21251 Ext. 265
Telex: 41030

THAILAND
UNIMESA Co., Ltd.
Elcom Research Building
2538 Sukumvit Ave.
Bangkok
Tel: 3932387, 3930338
Cable: UNIMESA Bangkok

UGANDA
Medical Only
International Aeradio(E.A.), Ltd.,
P.O. Box 2577
Kampala
Tel: 54388
Cable: INTAERIO Kampala

ZAMBIA
R.J. Tilbury (Zambia) Ltd.
P.O. Box 2792
Lusaka
Tel: 73793
Cable: ARJAYTEE, Lusaka

OTHER AREAS NOT LISTED, CONTACT:
Hewlett-Packard Intercontinental
3200 Hillview Ave.
Palo Alto, California 94304
Tel: (415) 493-1501
TWX: 910-373-1267
Cable: HEWPACK Palo Alto

CANADA

ALBERTA
Hewlett-Packard (Canada) Ltd.
11620A - 168th Street
EdmontonT5M 3T9
Tel: (403) 452-3670
TWX: 610-831-2431

Hewlett-Packard (Canada) Ltd.
210,7220 Fisher St. S.E.
Calgary T2H 2H8
Tel: (403) 253-2713
Twx: 610-821-6141

BRITISH COLUMBIA
Hewlett-Packard (Canada) Ltd.
837 E. Cordova Street
Vancouver V6A 3R2
Tel: (604) 254-0531
TWX: 610-922-5059

MANITOBA
Hewlett-Packard (Canada) Ltd.
513 Century St.
St. James
Winnipeg R3H 0L8
Tel: (204) 786-7581
TWX: 610-671-3531

NOVA SCOTIA
Hewlett-Packard (Canada) Ltd.
800 Windmill Road
Dartmouth B3B 1L1
Tel: (902) 469-7820
TWX: 610-271-4482 HFX

ONTARIO
Hewlett-Packard (Canada) Ltd.
1020 Morrison Dr.
Ottawa K2H 8K7
Tel: (613) 820-6483
TWX: 610-563-1636

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
Mississauga L4V 1M8
Tel: (416) 678-9430
TWX: 610-492-4246

QUEBEC
Hewlett-Packard (Canada) Ltd.
275 Hymus Blvd.
Pointe Claire H9R 1G7
Tel: (514) 697-4232
TWX: 610-422-3022
TLX: 05-821521 HPCL

FOR CANADIAN AREAS NOT LISTED:
Contact Hewlett-Packard (Canada) Ltd. in Mississauga.

CENTRAL AND SOUTH AMERICA

ARGENTINA
Hewlett-Packard Argentina S.A.
Av. Leandro N. Alem 822 - 12
1001**Buenos Aires**
Tel: 31-6063,4,5,6 and 7
Telex: 122443 AR CIGY
Cable: HEWPACK ARG

BOLIVIA
Casa Kavlin S.A.
Calle Potosi 1130
P.O. Box 500
La Paz
Tel: 41530,53221
Telex: CWC BX 5298,ITT 3560082
Cable: KAVLIN

BRAZIL
Hewlett-Packard do Brasil I.e.C. Ltda.
Avenida Rio Negro, 980
Alphaville
06400**Barueri** SP
Tel: 429-3222

Hewlett-Packard do Brasil I.e.C. Ltda.
Rua Padre Chagas, 32
90000-**Pôrto Alegre**-RS
Tel: (0512) 22-2998, 22-5621
Cable: HEWPACK Pôrto Alegre

Hewlett-Packard do Brasil I.E.C. Ltda.
Rua Siqueira Campos, 53
Copacabana
20000-**Rio de Janeiro**
Tel: 257-80-94-DDD (021)
Telex: 391-212-I905 HEWP-BR
Cable: HEWPACK
Rio de Janeiro

CHILE
Calcagni y Metcalfe Ltda.
Alameda 580-Of. 807
Casilla 2118
Santiago, 1
Tel: 398613
Telex: 3520001 CALMET
Cable: CALMET Santiago

COLOMBIA
Instrumentacidn
Henrik A. Langebaek & Kier S.A.
Carrera 7 No. 48-75
Apartado Aéreo 6287
Bogotá, D.E.
Tel: 69-88-77
Cable: AARIS Bogotá
Telex: 044-400

COSTA RICA
Cientifica Costarricense S.A.
Avenida 2, Calle 5
San Pedro de Montes de Oca
Apartado 10159
San Jose
Tel: 24-38-20, 24-08-19
Telex: 2367 GALGUR CR
Cable: GALGUR

ECUADOR
Calculators Only
Computadoras y Equipos Electrdnicos
P.O. Box 6423 CCI
Eloy Alfaro #1824,3 Piso
Quito
Tel: 453482
Telex: 02-2113 Sagita Ed.
Cable: Sagita-Quito

EL SALVADOR
Instrumentacion y Procesamiento Electronico de el Salvador
Bulevar de los Heroes 11-48
San Salvador
Tel: 252787

GUATEMALA
IPESA
Avenida La Reforma 3-48,
Zona 9
Guatemala City
Tel: 63627, 64786
Telex: 4192 Teletro Gu

MEXICO
Hewlett-Packard Mexicana, S.A., de C.V.
Av. Periférico Sur No. 6501
Tepepan, Xochimilco
Mexico 23, D.F.
Tel: 905-676-4600

Hewlett-Packard Mexicana, S.A. de C.V.
Ave. Constitucidn No. 2184
Monterrey, N.L.
Tel: 48-71-32, 48-71-84
Telex: 038-410

NICARAGUA
Roberto Terán G.
Apartado Postal 689
Edificio Terán
Managua
Tel: 25114, 23412,23454
Cable: RDTERAN Managua

PANAMA
Electrdnico Balboa, S.A.
P.O. Box 4929
Calle Samuel Lewis
Ciudad de Panama
Tel: 64-2700
Telex: 3483103 Curunda, Canal Zone
Cable: ELECTRON Panama

PERU
Compañía Electro Médica S.A.
Los Flamencos 145
San Isidro Casilla 1030
Lima 1
Tel: 41-4325
Cable: ELMEO Lima

PUERTO RICO
Hewlett-Packard Inter-Americas
Puerto Rico Branch Office
Calle 272,
No. 203 Urb. Country Club
Carolina 00924
Tel: (809) 762-7255
Telex: 345 0514

URUGUAY
Pablo Ferrando S.A.
Comercial e Industrial
Avenida Italia 2877
Casilla de Correo 370
Montevideo
Tel: 40-3102
Cable: RADIUM Montevideo

VENEZUELA
Hewlett-Packard de Venezuela C.A.
P.O. Box 50933
Caracas 105
Los Ruices Norte
3a Transversal
Edificio Segre
Caracas 107
Tel: 35-00-11 (20 lines)
Telex: 25146 HEWPACK
Cable: HEWPACK Caracas

FOR AREAS NOT LISTED, CONTACT:
Hewlett-Packard
Inter-Americas
3200 Hillview Ave.
Palo Alto, California 94304
Tel: (415) 493-1501
TWX: 910-373-1260
Cable: HEWPACK Palo Alto
Telex: 034-8300, 034-8493