



Mass Storage ROM

Manual Part No. 09845-93070

© Copyright Hewlett-Packard Company, 1981

This document refers to proprietary computer software which is protected by copyright. All rights are reserved. Copying or other reproduction of this program except for archival purposes is prohibited without the prior written consent of Hewlett-Packard Company.



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

Printing History

This manual is for use with the HP 9835A/B or the HP 9845B/C. It is a slightly revised version of the Mass Storage ROM manual, part number 09845-92070.

The changes which were incorporated into this latest edition are summarized in the System 45 Manual Revision Package (P/N 09845-93099). This package outlines the changes and additions that have been made to HP 9845 manuals.

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

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

April 1981...First Edition; Update Pages: ii, 1, 5, 6, 7, 11, 12, 13, 14, 15, 17, 19, 25, 26, 27, 28, 30, 41, 42, 50, 51, 70, 72, 73, 74, 77, 78, 82, 84, 85, 89, 91, 95, 96

December 1981...Second Edition; Update Pages: 5, 7, 12, 73, 84, 95, 95.1, 95.2, 96 through 116.

March 1982...Updated Pages: 5, 7, 73 and 95.2

July 1982...Third Edition: includes March 1982 Update.

January 1983...Fourth Edition

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Table of Contents

Chapter 1: General Information

Overview	1
Uses for Mass Storage	2
Buzzwords	4
Fundamental Syntax	5
Mass Storage Unit Specifier	5
MASS STORAGE IS Statement	8
File Names	8
File Specifier	10

Chapter 2: Getting Started

Initial Steps	12
Mass Storage ROM	12
Initialization	12
Interleaving	13
Flexible Disk Master-Slave System	16
Hard Disk Systems	16
Data Compatibility with Other Systems	17
Storage Operations	19
Types and Methods of Storage	19
Records	19
Files	21
File Directory	22
Cataloging to a String	25
Types of Files	27
Types of Access	29
Serial Access	29
Random Access	32
Creating Files	34
Record I/O	36
Writing Records	39
Print Verification	40
Reading Records	42
Using Serial and Random Access Together	45
Rapid Transfer of Arrays	45
Previewing a Data Item	48
User-Controlled End-of-File	51

Chapter 3: Storage Management

Fundamentals	54
Selecting Record Size	54
Overflowing Files	55
Copying Files	56
Purging Files	57
Special Operations	59
Protecting a File	59
Renaming Files	60
Execution from the Keyboard	60

Chapter 4: Data Transfers

Buffering	62
Device Buffering	62
Overriding the Device Buffer	64
Device Buffer Memory Requirements	65
Removing the Media	65
Additional Buffering	67
Conflict Between CHECK READ and BUFFER	68
Using Arrays as Buffers	68
Advanced Techniques	70
Passing Data Between Programs	70
Backup Files	72
Overlapped I/O	74

Chapter 5: Non-Data Files

Normal Usages	78
Storing Programs	78
Storing Key Definitions	83
Special Situations	84
Loading Binary Programs	84
Memory Snapshots	84
Effect of CHECK READ	85

Chapter 6: Errors and Error Processing

Hardware Errors	88
Hardware-Related Errors	88
What To Do About Hardware Errors	89
Anticipating Hardware Errors	90
Software Errors	91
Software-Related Errors	91
Anticipating Software Errors	91

Appendix A: Internal Tape Cartridge 93

Rewinding the Tape	93
File Directory	93
Tape Structure	94
Timings	94.1

Appendix B: Disk Drives 95

Interfacing	95
Initializing	96
Timings	96
Disc Drive Operating Characteristics	117

Chapter 1

General Information

Note

Before attempting to use high-speed disks (HP 7905M, 7906M/H, 7910H, 7920M/H, or HP 7925M/H) with your System 35 Desktop Computer, please contact your HP Customer Engineer to determine whether your system has the updated hardware that is necessary to support these disks.

Overview

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

The objectives for the manual are –

- To provide all necessary information to allow you to utilize compatible mass storage devices with your Series 9800 Desktop Computer.
- 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. It is recommended that quick reference be made primarily to the System 35 Operating and Programming Manual, or the System 45 BASIC Programming Manual and the Quick Reference provided with your computer.

It is assumed throughout the treatment that you are familiar with the basic operation and language used with Series 9800 Desktop Computers. 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, is 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 have an impact upon your programming. These are pointed out to you. Where there are differences between whole classes of devices – such as between tapes, flexible disks, and hard disks – they are also noted.

Particular operating, installation, and maintenance information for HP mass storage peripherals can be found in the operating or installation 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 B, and the device's operating manual.

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 your computer, or both.

Mass storage devices and media were developed for both of these reasons. Most computers, Series 9800 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 in 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 by offering a capability of storing large amounts of information in an easily accessible form. A program can 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 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 programs.
- 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.



Selection of the type of mass storage device to use is important to the functioning of your programs. Some applications run better on some types of devices than they do on others. In particular, for applications which involve heavy usage of mass storage files, such as non-consecutive file sorts and data base management, flexible disks or hard disks are recommended for optimum performance and reliability.

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

Mass Storage Unit Specifier

Many commands use what is known as a “mass storage unit specifier”, or **msus**. This specifier tells your computer 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 device type codes are shown in this table:

Device	M-Byte Storage Capacity	Device Type Code	9835 98331A ROM	9835 98331B ROM	9845 98413A/B/C ROM	Required Interface
Internal Tape	0.2	T	yes	yes	yes	n/a
7905M (Removable)	10	Y	no	yes	yes	98041
7905M (Fixed)	5	Z	no	yes	yes	98041
7906H (Removable)	10	C	no	yes	no	98041
7906H (Fixed)	10	D	no	yes	no	98041
7906M (Removable)	10	C	no	yes	yes	98041
7906M (Fixed)	10	D	no	yes	yes	98041
7908 (Fixed)	16	Q	no	no	**	98034 ¹
7910H (Fixed)	12	M	no	yes	*	98034 ¹
7911P (Fixed)	28.1	R	no	no	**	98034 ¹
7912P (Removable)	65.6	S	no	no	**	98034 ¹
7920H (Removable)	50	P	no	yes	no	98041
7920M (Removable)	50	P	no	yes	yes	98041
7925H (Removable)	120	X	no	yes	no	98041
7925M (Removable)	120	X	no	yes	yes	98041
9885M/S (Flexible)	0.5	F	yes	yes	yes	98032
9895A (Flexible)	2.3	H	no	yes	yes	98034 ¹

⌋

* 98413B/C ROM required.

** 98413C ROM required.

¹ No peripheral devices, other than mass storage devices may share this 98034 HP-IB interface. A maximum of two mass storage devices may share this interface.

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 B 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. 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 HP 9885 Installation Manual 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 ignored if included with an F device type or an M device type.

Some examples of mass storage unit specifiers:

If you want to specify the tape cartridge unit –

:T15

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 –

:Z9,0,0

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

:P1,0,0

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

:C implies :C12,0,0	:Q implies :Q7,0
:D implies :D12,0,0	:X implies :X12,0,0
:F implies :F8,0	:Y implies :Y12,0,0
:P implies :P12,0,0	:Z implies :Z12,0,0
:T implies :T15	:H implies :H7,0,0
	:M implies :M7,0,0
	:R implies :R7,0
	:S implies :S7,0

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 are all 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$[1;2]
"." & Type$
```

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 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

Many references are made, 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 medium 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, can 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 want to reference a file called “Backup” on a flexible disk with select code 10, you 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\$

produces 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**, your computer 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 tape cartridge unit).

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

Chapter 2

Getting Started

Page 13	INITIALIZE – enables a mass storage medium to have information recorded on it.
Page 22	CAT – generates a listing of the medium's directory.
Page 25	CAT TO – generates a listing of the medium's directory into a string array or R/W memory.
Page 35	CREATE – opens a file for data storage.
Page 36	ASSIGN – references a file by number.
Page 39	PRINT# – writes the data into the file on the medium.
Page 41	CHECK READ – verifies the data being read from a file.
Page 41	CHECK READ OFF – disables the CHECK READ statement.
Page 42	READ# – reads the data from the file on the medium.
Page 45	FCREATE – allows creation of data files for DMA-like array transfers.
Page 46	FPRINT – writes arrays at DMA speeds to files which are FCREATED.
Page 46	FREAD – reads arrays at DMA speeds from files which were FCREATED.
Page 49	TYP – identifies a data file as to its type.

Data File Type Codes

- 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 Not used
- 8 Partial string – beginning part
- 9 Partial string – middle part
- 10 Partial string – last part



File Directory Types

Type	Directory Abbreviation
Program	PROG
Data	DATA
Storeall	ALL
Keys	KEYS
Binary Program	BPRG
Binary Data	BDAT
Option ROM	OPRM
Assembly	ASMB
Data Base – Root	ROOT
Data Base – Backup	BKUP
Data Base – Data Set	DSET

Initial Steps

Mass Storage ROM

With the exception of the internal tape cartridge unit, no mass storage device can be used unless the Mass Storage ROM has been installed in your computer. The ROM required for your Series 9800 Computer is shown in this table.

System	ROM Required
9835	98331A or 98331B Mass Storage 1 and 98331B Mass Storage 2
9845B	98413A or 98413B or 98413C
9845C	98413A or 98413B or 98413C

There are two ROMs to be installed if you have the 98331B ROM with your System 35. They are called “98331B Mass Storage 1” and “98331B Mass Storage 2”. Both of these ROMs must be installed for your system to operate properly.

If you already have a 98331A ROM, remove it from your computer before installing the 98331B ROMs. Your computer will not operate properly if both A and B versions of the Mass Storage ROM are installed.

Refer to your System 35 Owner’s Manual or the System 45 Installation, Operation and Test Manual for ROM installation procedures.

Initialization

Before any particular mass storage device can be used with your computer, 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. 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, then you must “initialize” the medium itself.

Initialization is a process whereby your computer 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 your computer be initialized first. Each tape, flexible disk, and hard disk must first go through this process.

With tapes, the initialization process causes each physical record and inter-record gap to be created. 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. With hard disks, the check-out procedure was accomplished at the factory and the initialization only checks each track to see if it has been flagged as defective and replaced by a spare track.

From your point of view, the initialization process is a simple one and quite device-independent. From your computer's point of view, however, the effects of an initialization are quite diverse and are dependent upon the device and medium involved. All of your computer's concerns are bookkeeping, though, and your computer is the bookkeeper – not you.

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

```
INITIALIZE msus
```

(A select code must be specified in the msus parameter.)

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**¹ **only**, there is an additional parameter called the “interleave factor”. It can be added as follows –

```
INITIALIZE msus , interleave factor
```

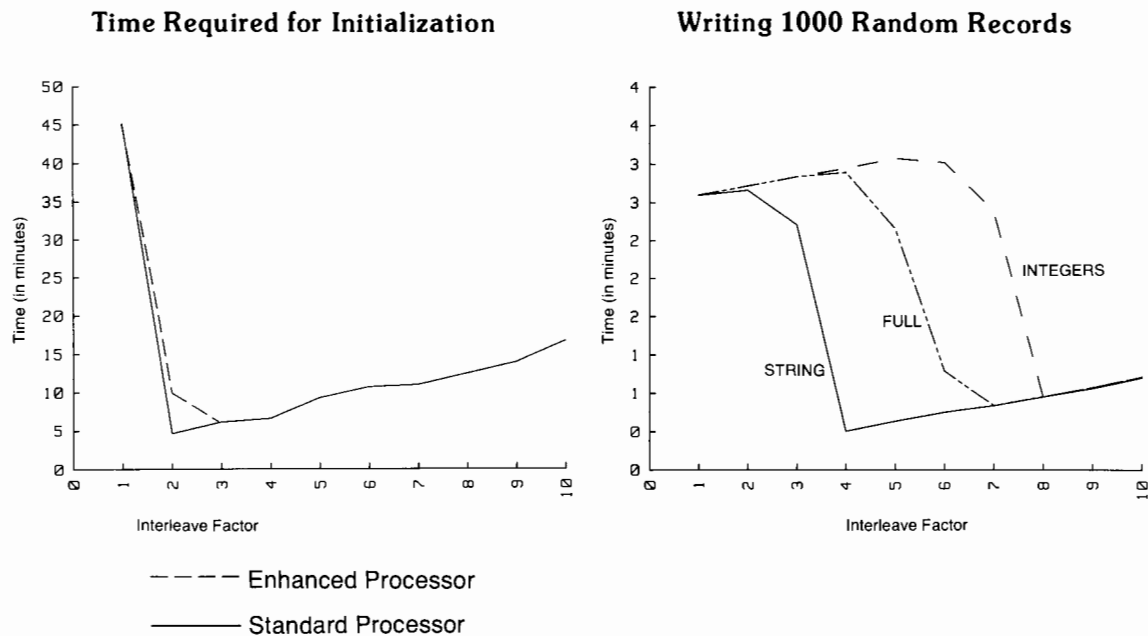
The purpose of this factor is to allow you to control the I/O efficiency of your disk. The **interleave factor** is a numeric expression which must round to an integer in the range of 1 through 10 when using the HP 9885 and in the range of 1 through 29 when using the HP 9895. If you omit it when initializing a flexible disk, it defaults to 7. If an interleave factor is used when initializing a tape or hard disk, the factor is ignored.

¹ For specific details on these drives, see Appendix B.

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, to transfer data to your computer, and to 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.

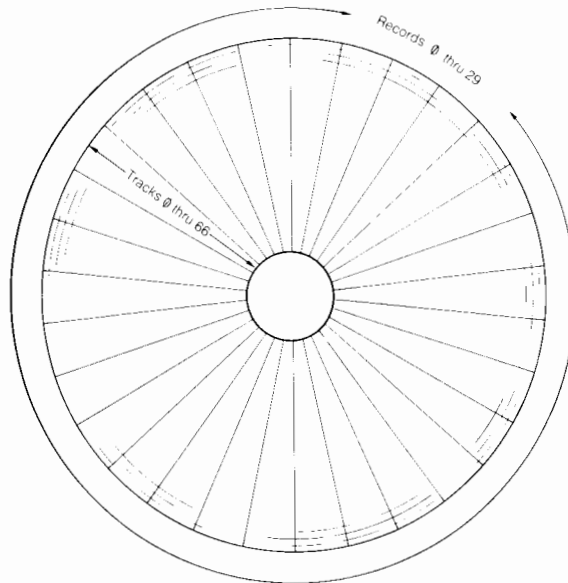
The following charts detail the 9885 disk performance under various interleave factors –



NOTE

When using the HP 9895A, the interleave factor should be set to 11 for increased speed in the transfers for strings, integers, and full-precision numbers.

Data are stored in concentric tracks on the disk. Each 9885 disk has 67 circular tracks, numbered 0 through 66. Each disk is also subdivided into 30 pie-shaped sectors. Each sector contains a number of records equal to the number of tracks (1 record = 256 bytes).

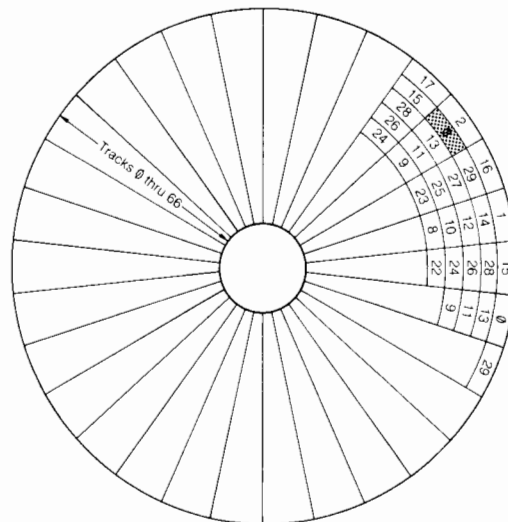


9885 Disk Structure

NOTE

Disks to be used with the HP 9885M/S Disk Drive should not be initialized on the HP 9895A Disk Drive. Because the HP 9895A initializes 77 tracks instead of 67, attempts to read data from the ten additional inner tracks (using an HP 9885M/S) may be unsuccessful. You may, however, use the HP 9885M/S Disk Drive to initialize disks that will be used with the HP 9895A.

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.



9885 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.

Hard Disk Systems

There are special considerations when initializing and using hard disk systems. Consult Appendix B before using a hard disk with the System 45.

Data Compatibility with Other Systems

Disks initialized for use on your Series 9800 desktop computer, and written under Series 9800 computer control can be used by other HP computers, such as the System 35 A/B, System 45A/B/C and even the HP 9831A. Data files are totally transferrable between the System 35 and System 45. Defining data files for transference with the 9831A requires that the record lengths must be 256 bytes (default length). Attempting to use files of different record lengths causes the 9831A to become lost on the disk. Series 9800 initialized disks cannot be used by the 9825A because of the lack of bootstraps required by the 9825. Disks initialized on the 9825A can be used on the System 45B/C, since the System 45B/C ignores the bootstraps.

Key files can be used between the System 35A/B and System 45B/C. Binary data files can be used with the System 35A/B and the System 45A/B/C.

Disks initialized for use on the 9825A, 9831A, 9835A/B, and System 45A/B/C may be used on your computer. Data files created by those machines may be used in every fashion the same as if they were created by your computer.

If you are using 9825- or 9831-initialized disks on your Series 9800 Desktop Computers, there are some precautions which should be observed –

- Both the 9825A and the 9831A may store values in a disk file which are outside the permissible floating point range of the System 35/45 (i.e., the absolute value is less than 10^{-100} or is greater than 10^{99}). Such numbers are ready by the System 34/45 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 200
120 IF A>9.999999999999999E99 THEN 300
130 ! Data are considered good...
```

are acceptable and work correctly, but the subsequent lines –

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

do 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 is unpredictable.

- Re-packing the file on the 9825A or the 9831A should not be done if the disk contains files of defined records which are less than 256 bytes. Files created by Series 9800 Desktop Computers should not be purged with the 9825A or 9831A if the record length is not 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 eleven kinds of files – **program**, **data**, **storeall**, **keys**, **binary program**, **binary data**, **option ROM**, **assembly**, and data base **root**, **backup**, and **data set** files.

Records

A **physical** record is the unit of storage dealt with by the mass storage devices themselves. They consist of 256 bytes each on Series 9800 Desktop Computers. 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 (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 2:1 or 3:1.

Logical records are a structured data concept and have no direct implementation in either the hardware or software of your computer. 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 to also 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 they appear 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 are 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



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.² Each file is in one particular location on the medium and all of its records are in order.³ 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	FILE #2	FILE #3
1st record	1st record	.
2nd record	2nd record	.
3rd record	3rd record	.
.	.	LAST FILE
.	.	
.	.	
Last Record	Last Record	

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.

¹ Provided the string doesn’t cross a defined-record boundary.

² The internal tape cartridge directory is used in a slightly different fashion than other media’s directories. See Appendix A for details.

³ 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.

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 45B 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 takes note of the location (track and physical record) of the first record of the file with that name. All other records in the file are 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.

File Directory

A directory, as mentioned above, is present on each medium (not the device). Replacing the medium in a device changes 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 –

NAME	PRO	TYPE	REC/FILE	BYTES/REC	ADDRESS
INIT		DATA	2	256	5
GARBAG		DATA	1	256	7
XDIAG		DATA	5	256	8
XWRITE		DATA	16	256	13
XREAD		DATA	14	256	29
LUNAR		PROG	46	256	43
LANDER		BPRG	44	256	89
XCORR		DATA	8	100	133
AWARD		PROG	7	100	141
REWRIT		ALL	5	256	148
EXPAND		DATA	8	80	153
HEAT01		BDAT	7	256	161
HEAT02		KEYS	1	256	171
DUMP		ALL	25	256	177
GARBAJ		DATA	1	256	210
BLKJK		DATA	1	256	425

Annotations in the original image:

- Circle around 'T15' in the NAME column, with an arrow pointing to 'Device Type'.
- Circle around '2' in the REC/FILE column, with an arrow pointing to 'Number of Tracks Available'.
- Circle around '256' in the BYTES/REC column, with an arrow pointing to 'Defined-record Size'.
- Circle around '5' in the ADDRESS column, with an arrow pointing to 'Physical-record Address of First Record in File'.
- Circle around 'INIT' in the NAME column, with an arrow pointing to 'File Name'.
- Circle around 'DATA' in the TYPE column, with an arrow pointing to 'File Type'.
- Circle around '2' in the REC/FILE column, with an arrow pointing to 'Defined Records Used'.

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 only lists 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 is 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
```

produces –

GARBAG		DATA	1	256	7
GARBAJ		DATA	1	256	210

The statements –

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

produce –

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 normally prints 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 statement in the above example had been –

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

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

Annotations in the image: An arrow points to the asterisk in the 'PRO' column for 'XWRITE' with the label 'File Protection'. Another arrow points to the '34' in the 'ADDRESS' column for 'XWRITE' with the label 'Track Address'. A third arrow points to the '14' in the 'ADDRESS' column for 'BLKJK' with the label 'Record 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.

Cataloging to a String – System 45 Only¹

Often a need arises necessitating the ability to treat a medium's directory as data. This can be accomplished by sending a catalog's output to a string array instead of to a printer. By selecting a string array which has been dimensioned to at least 41 characters per element, you can execute the statement –

```
CAT TO string array specifier
```

and have the resulting catalog output go to the string array specified.

For example –

```
DIM A$(1:10) [800]
CAT TO A$(*)
```



causes the individual file entries to be written to the individual elements in the array. They are written to the array starting with the first element in the array. Each entry appears in the element as it would if it were printed.

You can skip over a specified number of entries by including a skip value –

```
CAT TO string array specifier, skip value
```

skip value is a numeric expression representing the number of entries to be ignored before transferring entries to the array. Thus, if you were to say –

```
CAT TO A$(*), 2
```

then the first element in the array would contain the third file entry in the catalog. Skip values which are negative or zero have no effect.

Obviously, there can be no more file entries transferred to the array than there are elements in the array. If there are more entries than there are elements, then the additional entries are ignored after the array has been filled. If there are excess entries, then the entry number of the last entry actually transferred to the array can be returned in a numeric variable which you provide –

```
CAT TO string array specifier, skip value, return variable
```

If **return variable** is zero, then the last entry in the catalog has been transferred.

¹ The CAT TO statement described here is provided by the Advanced Programming ROM for the System 35.

This capability is particularly useful on media with large numbers of files. For example, suppose you are searching for three particular files on a medium with a considerable number of similarly-named files, then you could use the return variable to advantage –

```

10     DIM File$(3)[6],Catalog$(1:10)[80]
20     INTEGER File(1:3)
      .
      .
      .
100    MAT File=ZER
110    Skip=0
120 Catalog: CAT TO A#(*),Skip,Total
130    Skip=Total
140    FOR I=1 TO 10
150      FOR J=1 TO 3
160        IF POS(A#(I),File$(J)) THEN File(J)=1
170      NEXT J
180    NEXT I
190    IF File(1) AND File(2) AND File(3) THEN Found
200    IF Skip THEN Catalog
210    DISP "Required files not found"
      .
      .
      .
300 Found: DISP "All required files present"
      .
      .
      .

```

You may specify any of the usual catalog items described in the sections above –

- selective catalog specifier
- msus
- heading suppression

This is done in the same fashion as above, by making the statement appear as –

```

CAT [TO string array specifier [, skip variable [, return variable] ] ]
    [; selective catalog specifier / msus [, heading suppression] ]

```

The **selective catalog specifier** and **msus** work the same as above. **heading suppression** works in somewhat the opposite fashion from the above; it is a numeric expression, but if it evaluates to any other value than 1, then it transfers the “heading” of the catalog to the first element of the array and the entries begin with the second element. This “heading” consists of the msus and the number of valid tracks on the medium. If the expression evaluates to 1, then this heading is suppressed and the first element in the array is a catalog entry, as described above.

Finally, if a **selective catalog specifier** is included, only those entries which conform to the specifier are transferred to the array, as might be expected. It should be noted that the return variable contains the actual entry number of the last entry transferred, regardless of the entries rejected because they don't conform to the specifier. Note, then, the effect on the efficiency of the above example if you changed the example as follows –

```
119 Search#=File#(1)[1,4]
120 Catalog: CAT TO A#(*),Skip,Total;Search#
```

Of course, this only works if all the files being searched for start with the same four characters.

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 many types of files –

Type	Directory Abbreviation
Program	PROG
Data	DATA
Storeall	ALL
Keys	KEYS
Binary Program	BPRG
Binary Data	BDAT
Option ROM	OPRM
Assembly	ASMB
Data Base – Root	ROOT
Data Base – Backup	BKUP
Data Base – Data Set	DSET

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 characteristics –

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# instructions. Used to store programs in “source” form or data written by PRINT# instructions.

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 instructions. Used by DMA mass storage devices (but not the internal tape cartridge) to do fast data transfer at DMA speeds.

OPTION ROM, ASSEMBLY, DATA BASE – Created by the option ROMs available on your computer. These files are used by the option ROM creating them. For details on their use, consult the manual for the ROM involved.

If you are cataloging a medium which was initialized on a 9825A or a 9831A, the computer may not be able to determine accurately the file type. In such files where it is not positive about the type, your computer attempts a determination and the catalog output contains that determination, along with a question mark. In some cases, particularly with the program file type (due to different operating systems), your computer may not be able to use the file as originally intended. For example, here is a directory of a tape cartridge showing non-compatible programs.

NAME	PRO	TYPE	REC/FILE	BYTES/REC	ADDRESS
T15			2		
MAZE		PROG?	74	256	5
STAR		PROG?	10	256	91
BABCHR		PROG?	54	256	101
BINARY		PROG?	75	256	155
index		PROG?	54	256	230
SORTER		DATA	14	256	284
AMORT		PROG?	5	256	298
xmpl		PROG?	2	256	304
MERGE		PROG?	30	256	359
LOGIC		PROG?	36	256	389
+++++		DATA	1	256	425
INDEXR		DATA	62	256	426
Trek		PROG?	212	256	488
EXAMPL		DATA	100	256	700

Types of Access

There are two methods (or “modes”) of data access with your computer – 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.

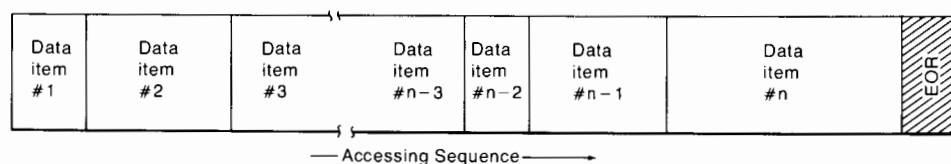


Serial Access

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 straight the location of things.

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 –

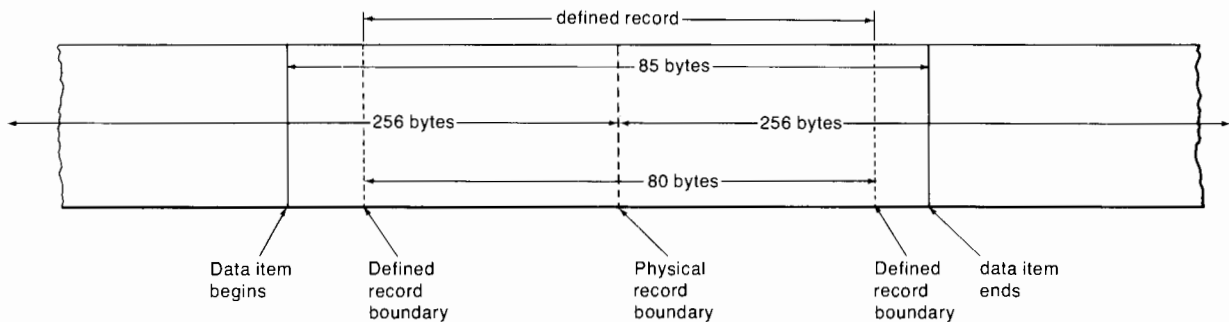


As data is written in a serial file, previous end-of-file (EOF) marks which may have existed there are written over as they are encountered. An exception to this is made when the EOF mark at the beginning of a defined record would be written over by an EOR. Instead of doing this, the EOF mark is allowed to remain. The file pointer is still positioned there, however, and the next serial write treats the EOF as an EOR and writes 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 write a serial access record of 85 bytes to a file which has defined records of 80 bytes each, then obviously it takes more than one defined record to hold the information. Therefore, you must pass over at least one defined-record boundary. The following diagram demonstrates what this might mean –



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

Serial access is sequential in nature. When you write a logical record, as 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 is permitted to remain in the file. This occurs whenever the next item 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 is left there and the number is 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 remains and the string begins with the next record.¹ If more than four bytes remain, it puts as many characters as it can in the record (there is an overhead of four bytes), and continues the string in the next record, if necessary.

You must be careful in the selection of defined-record sizes. Making a poor selection of record size can significantly waste storage space. For example, if you chose a defined-record size of 14 and write serially a number of full-precision numbers (8 bytes each), with nothing in between, you are wasting 6 bytes per record. This effectively wastes 43% of the storage space in the file. Thus, it is 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 ignores EOR marks left in the file, so you do not, as a programmer, have to take them into account in your serial reading.

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.

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 lengths.
- Data is a sequence of items with markers separating them, i.e., data is read and written sequentially.
- The last data item written is followed by an EOC mark. This mark is destroyed by the next data item written at this point unless there is a failure record in the defined record for that item.
- Defined record boundaries and physical record boundaries, as well as EOCs in a file, may be ignored for most programming purposes.
- Defined record size affects storage efficiency and program performance.
- It is an 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 there is 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
- It is a good method with disks, not so good with long files on tape cartridges

Creating Files

As was pointed out earlier (page 27), 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”.

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 is rounded up to an even number. If **record length** is omitted, the defined-record length for the file defaults 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 rounds to an integer between 1 and 10.

return variable may be any numeric variable. After execution of the ASSIGN, the return variable contains a value indicative of the status of the file, thus indicating if an error occurred 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 is an error. There is also 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 is first assigned to Keep1, then to Keep2, and finally to Keep3. Each reassignment cancels out the one before it. This happens upon **execution** of the ASSIGN statement. Thus, if after line 1000, the program loops back to 100 (say by a GOTO 100), #1 is 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 are the same as references to different files. Thus, with this facility, you can 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, 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 causes 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 causes 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 will cause 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.

Executing a STOP or END statement causes all open files to be automatically closed.

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.

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** move 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 your computer 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 tries 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 on the ON ERROR statement can be found in the BASIC Programming Manual.) The ON ERROR trap does not work on CHECK READ if you are processing in the OVERLAP mode (see “Overlapped I/O”, pages 73-74).

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 is 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 38), 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 be verified. Directory updates (caused by COPY, STORE KEY, CREATE, FCREATE, PURGE, PROTECT or RENAME) are also checked when the CHECK READ is in effect. 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 necessarily enables 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 BASIC 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 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 –

numeric	numeric	numeric	numeric	string	numeric	string	string	numeric
---------	---------	---------	---------	--------	---------	--------	--------	---------

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 44), 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 is 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$
```

is perfectly valid for this record, but –

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

does not work at all.

MS-44 Getting Started

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 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 between the device and the memory.

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 –

$$\text{INT}(\text{array length} / 256) + \text{INT}(\text{array length} / 65536) + 3$$

where **array length** = number of elements × (bytes per element + 2) when the **bytes per element** is the **dimensioned** length of each element (**not** the current string length of the individual elements). Add one additional byte per element if the dimensioned length is odd.

- For numeric arrays,

$$\text{INT}(\text{array length} / 256) + \text{INT}(\text{array length} / 65536) + 3$$

where **array length** = number of elements × k with:

k=2 if an integer array

k=4 if a short-precision array

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 –

```
FFPRINT file specifier , array identifier
```

and to read such an array the statement is –

```
FFREAD 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 is 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 for a demonstration of this effect.

String arrays are also 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 simply writes 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 are **not** verified via the CHECK READ which may be in general effect (see page 40). 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  OPTION BASE 1
20  DIM A(5,20),B(10,10),C(10,5),D(10,5)
30  File$="RANDOM:F"           ! Create BDAT file big
40  FCREATE File$,100*8/256+2 ! enough to hold A(*)
50  RANDOMIZE
60  FOR I=1 TO 5
70      FOR J=1 TO 20
80          A(I,J)=RND
90      NEXT J
100 NEXT I
110 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”).

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.

BDAT files created by System 35A/B or System 45A may be FREAD by a System 45B/C, but they may not be FPRINTed.

Previewing a Data Item

One requirement for any READ#, be it serial or random, is that the data types of stored data and the variables into which they are being read correspond. On some occasions, you may not know in advance what the data type of an item is. In such cases, you should find out the data type before doing the READ# and then 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 is the datum which is 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 is **not** an EOR, but is 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 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 –

```
10  INTEGER I
20  SHORT S
30  DIM File#[10]
.
.
.
200 ASSIGN #1 TO File#
210 FOR J=1 TO 10
220  ON TYP(1) GOTO 230,250,270,270,310,290,270,250,250,250
230  READ #1;F
240  GOTO 220
250  READ #1;A#
260  GOTO 220
270  DISP "ERROR IN POSITION ATTEMPT"
280  STOP
290  READ #1;S
300  GOTO 220
310  READ #1;I
320  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 will write a single EOF mark to the file at this point in the file, writing over the EOR mark which is usually 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.

There can be more than one EOF mark in a file, but there cannot be more than one EOF mark in each logical record. This allows random access to create and detect records with less data than the maximum allowable for each logical record.

Notes

Chapter 3

Storage Management

- Page 56 **ON END** – used to branch the program when an End-of-file is encountered.
 Page 56 **OFF END** – disables the ON END statement.
 Page 56 **COPY** – copies a file to a mass storage medium.
 Page 57 **PURGE** – removes the file from the mass storage device.
 Page 59 **PROTECT** – adds a password to the file name.
 Page 60 **RENAME** – changes a file name.

```

1  REM *****
2  REM ***
3  REM ***
4  REM ***           Media Copy Program
5  REM ***
6  REM ***
7  REM *****
10 DIM Catalog$(1:2)[41],File#[6]      ! Strings to receive data
20 Media:! Finds out which media are involved
30 LINPUT "What is the MSUS of the medium to be copied ?",Old$
40 Old$=UPC$(TRIM$(Old$))              ! Cleans up the input string
50 IF Old$[1,1]<>":" THEN Old$=":"&Old$  ! Appends colon for MSUS
60 LINPUT "What is the MSUS of the target medium ?",New$
70 New$=UPC$(TRIM$(New$))             ! Cleans up the input string
80 IF New$[1,1]<>":" THEN New$=":"&New$  ! Appends colon for MSUS
90 IF Old$<>New$ THEN Copy_files      ! Copy if different media
100 DISP "Shouldn't copy to the same medium --";
110 GOTO Media
120 Copy_files: ! Copies the files in pairs
130 ON ERROR GOSUB Duplicate_name      ! Error checking
140 CAT TO Catalog$(*),Counter,Counter;Old$ ! Cat to string array
150 FOR Loop=1 TO 2                    ! Initializes loop
160 IF NOT LEN(Catalog$(Loop)) THEN STOP ! Stop when done
170 File#=Catalog$(Loop)[1,6]         ! Obtains the file name
180 DISP "Working on <;File#;>"       ! Operator update
190 IF Catalog$(Loop)[15;1]<>?" THEN COPY File#&Old$ TO File#&New$
200 NEXT Loop                          ! Repeats loop
210 IF Counter THEN Copy_files        ! Repeat for more copies
220 STOP
230 Duplicate_name: ! Error checking for duplication of names
240 BEEP
250 IF ERRN<>54 THEN Fatal_error       ! How to die gracefully
260 DISP "FILE <;File#;> already exists on ";New$
270 RETURN
280 Fatal_error: ! Unrecoverable at this point
290 DISP ERRM#
300 END

```

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 in combination 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 35-36). 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 maximizes the efficiency of the system's mass storage routines. However, while the defined-record size may not be so important in such cases, the file size is. It is necessary that your file be large enough to hold the data you are 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 are 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 is selected that is too small to handle all of your data, you must create another file to handle the overflow. Then you 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 –

```

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$(Counter)&Device#
100 CREATE Filename#,Records      ! Create and assign
110 ASSIGN #1 TO Filename#        ! the first file
120 ON END #1 GOSUB Newfile       ! When file's end is
130 !                             reached, create another
140 !   Program follows
150 !
   .
   .
   .
500 PRINT #1;...
   .
   .
   .
1000 !
1010 ! File-creation routine follows
1020 ! Creates files by incrementing a counter and using it
1030 ! as part of the file
1040 !
1050 Newfile: Counter=Counter+1
1060 Filename#=#File#&VAL$(Counter)&Device#
1070 CREATE Filename#,Records      ! Defaults to
1080 ASSIGN #1 TO Filename#        ! physical-record size
1090 PRINT #1;...
1100 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 causes “Newfile” to be performed which creates a file called “DATA2”, assigns it to #1, and prints the information to it instead of to the first file. This continues until the program is finished, creating files each time the previous one is filled. Upon completion, there is stored on device “:F” (the flexible disk drive), one or more files called “DATA....” After the program is finished a CAT “DATA:F” reveals the directory data on the files created.

The ON END statement that is 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 “Overlapped I/O in Chapter 4”).

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 to 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 is identical to the first – same file type, record size, number of records, and contents. Both files then exist; the old one is not purged.

Any kind of file may be copied with this statement, not just those of the DATA-type. See also “Protecting a File” (page 59) 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 erroneous 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 "repacked" –

```

10  MASS STORAGE IS ":T"
20  ON ERROR GOTO Error
30  Temp$="TEMP:F"
40  FOR I=1 TO 13
50  File$="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 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 fit, the files are written one immediately following the other. Progressively, then, the gaps between files collect 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 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-string¹), 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 do not work unless the ASSIGN to the file has been made with the proper protect code (see page 36). However, a GET statement, all other things being equal, is permitted, since it only **reads** the file.

¹ A null-string will be interpreted as “no protection”.

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 has the same protect code as the old file.

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 is 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 identifier  
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** does 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.

Chapter 4

Data Transfers

- Page 67 **BUFFER** – allocates a buffer for an assigned file.
Page 74 **OVERLAP** – allows simultaneous I/O and processing.
Page 75 **SERIAL** – disables simultaneous I/O and processing.



Device Buffers Summary

Maximum Buffer Size	256 bytes
Maximum Number of Buffers	4 (one per each select code)

Special Considerations

- Data transfer can only be through the buffer.
- Currently unused buffers are de-allocated when storage space is needed for other system requirements.
- Flushing of the buffer occurs whenever –
 - Buffer is full (on a PRINT#)
 - Buffer is empty (on a READ#)
 - Access is made to a different file on the same device
 - Access is made to a different device on the same select code
 - READ# follows a PRINT#
 - PRINT# follows a READ#
 - PRINT# to a file with CHECK READ
 - Program executes a STOP, END or PAUSE
 - Keyboard execution of PAUSE or STOP
 - Buffer has been de-allocated for storage reasons

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 your computer 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 your computer. 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) is written to the **buffer** for the device, and not directly to the device itself. Since A(I) is a full-precision number, eight bytes are written to this buffer each time PRINT# is executed. After 32 iterations all 256 bytes of the buffer are filled. Then the entire contents of the buffer are physically written to the device, the buffer is erased, and it starts to fill all over again. This is called “flushing” the buffer. Each time the buffer fills, it automatically flushes 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 also occurs whenever a STOP or END statement is encountered, and whenever PAUSE or STOP is entered from the keyboard. The buffer for a particular file is also 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 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 are 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 your system. Each mass storage device accesses 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

```

has a vastly improved execution performance if the statements are reorganized so that access is 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 there is 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. Thus the value of having a buffer is 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 the 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 acceptable one. For example, in a data acquisition application, data being recorded every 30 minutes might take up to 16 hours to fill a buffer. The risk of something going wrong and losing **all** of the data during a long period of time is proportional to the time that the data is being held in the buffer. In such applications it may be best to force what is called an “immediate-write” on a file.

An immediate-write for a file flushes the buffer on its mass storage device after every PRINT# operation. This capability is a consequence of a CHECK READ being in effect for the file (see pages 40-41).

Thus, to override the internal buffering on a device, use the CHECK READ statement for the file concerned to force an immediate-write after every PRINT#.

There is no need to override the buffering with regards to READ# operations.

Device Buffer Memory Requirements

Buffers come out of the system's own memory space. The system has room for four buffers and for each buffer employed there are 256 bytes of memory allocated to it.

Buffers are allocated by your computer's operating system to select codes as the devices are needed. If there is no further room for a buffer in available system memory because of other system needs, then the system will flush and de-allocate a buffer not currently in use. It determines this by ascertaining whether or not a PRINT# or READ# is currently active for the device.

If there are more select codes active than there are available buffer areas, then a buffer is re-allocated each time reference is made to a device on a select code without an active buffer. Since this kind of juggling can affect program performance in the OVERLAP mode (see "Overlapped I/O", in this chapter), if you have more than four select codes active at one time, then you would be wise to arrange the execution of your I/O statements to minimize the switching of execution between files on different select codes. Try instead to execute as many I/O operations as possible to a given file before switching operations to another.

Removing the Media

Removing a tape cartridge or disk before an open file has been closed can cause some data to be lost. This is because the device buffer may not have been flushed. To avoid this circumstance, a medium with open file should remain undisturbed in the device until one of the following occurs –

- The open files are explicitly closed with ASSIGN * statements.
- A STOP statement is executed.
- An END statement is executed.
- A PAUSE statement is executed.

Device Buffers Summary

- There is one 256-byte buffer in system memory for each select code connected to the system (maximum of 4) and used by a program.
- Data transfers occur only through the buffer.
- Flushing occurs whenever —

A buffer is full (on a PRINT#) or exhausted (on a READ#);

Access is made to a different file on the same device;

Access is made to a different device on the same select code;

A READ# follows a PRINT# or vice versa;

A PRINT# to a file with a CHECK READ enabled;

A STOP, END, or PAUSE is executed from a program, or a PAUSE or STOP is executed from the keyboard;

The buffer is de-allocated for storage reasons.

- Currently unused buffers are de-allocated when the system requires storage space for other purposes.
- Media with open files should not be removed from the device until the buffer has been flushed.

Additional Buffering

As a user, you have little control over the device buffers. There are many imaginable circumstances where the suggested kind of efficient program modification is not possible. 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
140 B=A*C(I)/D
150 PRINT #2;B
160 NEXT I

```

is not amenable to rearrangement as was the example on page 63, at least not without using significant additional memory space. Thus, it might not be reasonable even to try to do so. But if you still want to gain the advantages of buffering, you can get them by employing the BUFFER statement.

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 a similar fashion to 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

```

you achieve the same kind of savings you 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 36-37). 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 is 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-write”.

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 still performs its “read-after-write” function, but only after the file buffer has been filled and is flushed. It does 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 BASIC 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 language is structured, faster still are the FPRINT and FREAD statements for non-tape media. For details on these statements, see pages 45-48.

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 avoid 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 48-49) to assure that items are of the proper data type.
2. Use random access mode whenever feasible. This permits you to better control 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 causes 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 causes 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 a 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

It is extremely important to have backup copies of the programs and data files which are on your disks. If read data errors should occur on one of your disks (or if a hardware failure should occur), information can be lost. Therefore, a duplicate copy of each of your files decreases the chances of your data becoming permanently lost.

Your backup files should always be on a different medium, so if anything happens to one medium you still have the other to use. If you have only one disk drive and the disk is removable, you can create a backup using the program below. The procedure is quite time-consuming, however. If you have large amounts of valuable information that should be backed up frequently, the use of a second disk drive is strongly recommended.

Backups are easy to create through the COPY statement (see page 56 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 creates 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.

There are two methods for backing up your data: selective backup and media backup. Selective backup involves copying only a portion of your files, since some files may be more valuable to you than others. Copying all of the files from your disk to another medium is called media backup. Whether you do selective or media backup, you should do it periodically (depending on the value of your data), and especially after making changes to your data.

Some media may be more suitable for backing up your particular disk. The following table shows which media are appropriate (*) and most appropriate (**) for media backup.



BACKUP MEDIA

ORIGINAL MEDIA	M-byte storage capacity	inter- nal tape	9885 disk	9895 disk	7905 rmvbl disk	7906 rmvbl disk	7908 cart. tape	7910 disk	7920 disk	7925 disk	7970E Opt. 826 mag. tape
internal tape	0.2	**	*	*	*	*		*	*	*	*
9885 disk	0.5	*	**	*	*	*		*	*	*	*
9895 disk	1.2		*	**	*	*		*	*	*	*
7905 fixed disk	5				**	*		*	*	*	**
7906 fixed disk	10					**		*	*	*	**
7908 disk	16						**				
7910 disk	12				*	**		*	*	*	**
7911 disk	28.1						**				
7912 disk	65.8						**				
7920 disk	50								**	*	**
7925 disk	120									**	**

The appropriate backup media for the HP 7908 is the built-in cartridge tape. With extra effort you may also backup a 7908 disk to a 7920 disk, a 7925 disk or the HP 7970E (Opt. 826) magnetic tape.

For selective (single file) backup, any medium is appropriate as long as it is large enough for the file you want to copy. You should use the COPY statement for this type of backup.

HP's Hard Disc Utilities Package is available to facilitate media or selective backup from one disk drive to another of the same type. If you want to back up a disk (all unprotected files) to another type of media, use the following program:

```

10  ! Use this program for media backup between like or unlike devices.  It
20  ! copies all unprotected files from your original media in groups of five.
30  !
40  Original_msus$=":"
50  Backup_msus$=":"
60  !
70  !
80  EDIT "What is the msus for your original disc or tape?",Original_msus$
90  MASS STORAGE IS Original_msus$
100 EDIT "What is the msus for your backup disc or tape?",Backup_msus$
110 !
120 OPTION BASE 1
130 DIM Catalog$(5)(80)           !Dimension Catalog$ to hold five
140 !                             entries of a catalog listing.
150 Ret_var=0
160 !
170 Copy_loop: Next_file=Ret_var   !Ret_var identifies next file entry
180 !                             in Catalog listing.
190 CAT TO Catalog$(*),Next_file,Ret_var !Catalog$ contains five entries of
200 !                             catalog listing.
210 FOR I=1 TO 5                 !Copy files in groups of five.
220     IF Catalog$(I)="" THEN Check !If there are no more files to
230     !                             copy, then check Ret_var.
240     IF Catalog$(I)(8,1)="" THEN GOTO Next_entry !Protected files are not
250     !                             copied.
260     COPY Catalog$(I)(1,6) TO Catalog$(I)(1,6)&Backup_msus$ !Copy one file.
270     !
280 Next_entry: NEXT I
290 !
300 Check: IF Ret_var<>0 THEN Copy_loop !When Ret_var=0, there are no more
310 !                             files to copy.
320 DISP "Backup is complete."
330 STOP
340 END

```

Overlapped I/O

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

It is quite possible for your computer 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 your computer outputs these values, it goes back and does the next three calculations, and so on. So actually, your computer 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 also disables the OVERLAP mode. If an ON END statement is in effect, OVERLAP necessarily is disabled for that file. This effect occurs because of the intended purpose of the ON END statement. The ON END assumes that some branch – and alternate execution sequence – takes 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 partially cancels the effect of an OVERLAP. After executing an OFF END for a file, you automatically go back to overlapped processing. For a discussion of the effects on programming methods, see “Overflowing Files” in Chapter 3 (page 55).

The ASSIGN, FPRINT, and FREAD statement 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.

Notes

Chapter 5

Non-Data Files

- Page 78 **STORE** – records a semi-compiled program and binary program(s) from R/W memory the first time the file name is used.
- Page 78 **RE-STORE** – replaces a semi-compiled program and binary program(s) from R/W memory with another version having the same file name.
- Page 78 **LOAD** – enters the semi-compiled program and binary program(s) into R/W memory.
- Page 79 **SAVE** – records the source code program from R/W memory the first time a file name is used.
- Page 79 **RE-SAVE** – replaces the source code program from R/W memory with another version having the same file name.
- Page 80 **GET** – enters the source code program into R/W memory.
- Page 83 **STORE KEY** – records the SFK definitions into a special file.
- Page 83 **LOAD KEY** – enters the SFK definitions from a special file.
- Page 84 **LOAD BIN** – enters the binary program(s) from a file.
- Page 84 **STORE BIN** – records the binary program(s) into a file.
- Page 84 **STORE ALL** records a memory snapshot into a file.
- Page 84 **LOAD ALL** – enters the memory snapshot from a file.

Most mass storage operations focus upon data files. However, there are five other types of files available for your use. They each have particular attributes which make them useful in certain applications.

Normal Usages

Storing Programs

Obviously one of the primary uses of mass storage is to store **programs**, as well as data. It does you little good to be writing general application programs if you have to enter them from the keyboard every time you want to use them.

There are three ways to store a program on mass storage – store it in its quasi-compiled form, save it in its source-code form, or create it by another program.

To store a program in its quasi-compiled form, it is first necessary to have the program in memory. Then the statement –

```
STORE file specifier
```

will take the program along with all binary routines in memory (i.e., the compiled version), plus all of the necessary symbol tables, and store the lot of it in the file specified. This statement is best used from the keyboard, but can be executed from a program as well.

If you want to replace a version which already exists on the medium with the same name, then the statement to use is –

```
RE-STORE file specifier[ ,protect code]
```

A file used in the STORE or RE-STORE commands is known as a “PROGRAM-type” file.


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.

NOTE

A program which includes enhanced or color graphics keywords and has been STOREd on a 9845B Model 2XX or a 9845C cannot be LOAded on a 9845B Model 1XX.

If you want to immediately execute a program after loading it, you can either use the LOAD statement as above and then press the  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
```

* LOAD should not be executed when TOPEN is active. Refer to the I/O ROM Manual for information about TOPEN.

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, variables and binary programs 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 (except that a protect code may be included when necessary) –

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

Execution of the RE-SAVE statement first purges the old copy and then saves the new one. If the new one is larger than the old it is possible to get a “medium overflow” error (number 64) when the new copy is attempted. The result of this is that the old copy is gone, but the new copy is not saved. Another medium must be used in such cases.

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 execute 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
```

becomes –

```
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 is that from the file. All lines of the original program after line 30 were 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 begins 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 causes 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 commences at the line indicated. If you do not specify an execution line, then execution proceeds 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 immediately begins 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 –

```
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 PRINT #1;"101 END"
170 ASSIGN * TO #1
180 GET "RANDOM"
190 END
```

This program creates the file and stores in it 100 generated strings which themselves make up another program. The GET statement will fetch this program. When a GET is executed from a program, the new program just retrieved is automatically executed, so this program, in the end, prints 100 random numbers as well as changes 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** are 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 to use is –

```
STORE BIN file specifier
```

The statement causes 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 are 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 then **becomes** the machine state.

Note

In order to LOAD ALL a STORE ALL file, your computer must be identical (options and memory size) to the one used when the STORE ALL was executed.

There is an exception to this which modifies the actual machine state somewhat. All file and device buffers are flushed and the file tables are erased. This means that all assignments to file numbers (ASSIGN statement) are cancelled, and must be re-done. Similarly, ON ENDS are cancelled, as well as CHECK READ for all files. These things must also be restored, if desired.

Note

If your 9845 is equipped with a light pen, execute GRAPHICS INPUT IS OFF before attempting a LOAD ALL or STORE ALL operation.

This type of file can be used for those larger applications where periodic checkpoints are desired in case they are needed to recover from an error which occurs after the snapshot is taken. By using the STORE ALL file, you can return yourself essentially to some point before the error and proceed from there, either to re-duplicate the error, or just to recover and get the desired results by closer monitoring. Here is just such an example –

```

20  CREATE "EMDATA",40          ! Create file for employee data.
30  ASSIGN #1 TO "EMDATA"
40  PRINT #1,1;223098756,2290876,9344,49
50  PRINT #1,2;107490349,2260967,2113,28
.
.
.
.
.
430 PRINT #1,40;548119087,2273993,7439,25
440 READ #1,1                  ! Move file pointer to first
450                             ! defined record.
460 Check data: FOR I=1 TO 40
470 READ #1,I                  ! Read each record.
480 PRINT Idno,Ssno,Phone,Age
490 IF Age>55 THEN PRINT "*** OVER 55. ***"
.
.
.
.
.
600 STORE ALL "BCKUP"&VAL$(I)
610 NEXT I
620 END

```



Should anything happen during the execution of this program, consult the catalog for the mass storage device being used and find the last BCKUP file. By Executing a LOAD ALL on that file, and re-ASSIGNing file #1 to "EMDATA", you would either duplicate the error under your supervision or just recover and get the results you wanted all along.

NOTE

STOREALL and LOADALL are not permitted when a sub-program (SUB or multi-line function) is executing.

Effect of CHECK READ

If you have executed a CHECK READ without a file number (see Print Verification), then all writing operations to every device are verified. This includes the STORE, RE-STORE, SAVE, RE-SAVE, STORE KEY, STORE BIN, and STORE ALL operations. Since verification slows the I/O process and some of these statements are used for their speed and efficiency, it may be undesirable to leave the CHECK READ in effect while executing these operations. Verification can be disabled with the CHECK READ OFF statement.

Notes

Chapter 6

Errors and Error Processing

Overview

This chapter discusses the following materials and techniques:

- Hardware errors, and how to deal with them.
- Software errors, and how to deal with them.

Errors which occur in the use of mass storage can generally be classified into two groups – errors which are hardware-related and those which are software-related. Actually, there may be some overlap between these two groups in a given situation. What distinguishes them is the action which must be taken to correct a problem and to allow processing to proceed.

It is intended that this chapter give some guidance as to how certain errors can be handled. It is not a definitive checklist of what can go wrong, nor is it a thorough treatment of the means to correct the difficulties which are listed. Rather, it is a reference on **some** of the things which can go wrong, what might cause them, and how to deal with them. Each programmer has a unique method of approaching the problem of error processing, and there is no way to anticipate all of them. Even so, the following should offer some assistance in identifying the source of an error.

Hardware Errors

Hardware-Related Errors

There are four hardware conditions which can create errors: power loss, equipment failure, cable separation, and media wear. Since some of these conditions can create software errors, check to see that one of the following is not at fault before pursuing a software error.

- **Power loss.** Either because of power cord separation, or general power interruption, your computer, or the mass storage units, or both, may lose power. Should it be the computer, all information in the memory is lost and when power returns, the initial power-on conditions (and defaults) prevail. You lose both your program and your data. If you have an unmonitored program running and you return to find the unit with the power on, but with no program, then you probably experienced a temporary power interruption.

If power is lost to an 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
```

is displayed and the system pauses and periodically queries (“polls”) the missing (or malfunctioning) unit (“nn” is the number of the select code where the problem exists). It continues this way until the unit is brought on-line, at which time the command –

```
READY # nn
```

should be executed from the keyboard. The system polls the unit one more time and then allows 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) may appear.
- **Cable separation.** The desktop computer may lose communication with external mass storage devices should the cable connecting them malfunction. It definitely loses 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 begins 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 9835 System Exerciser Manual or the System 45 Installation, Operation and Test Manual).

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 does not work sufficiently to even allow the test to be performed, then call an HP Sales and Service Office.

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 is the case, you probably have 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.

Hardware-related mass storage errors; 19, 57, 65, 66, 69-76, 80-90.

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 BASIC 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.

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.

Software-related mass storage errors; 16, 20, 38, 39, 46, 50-56, 58-64, 67, 68.



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 BASIC Programming Manual contains the information you need to use 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



Internal Tape Cartridge



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  has been assigned REWIND“:T14”. Key  has been assigned REWIND “:T15”. Pressing either of those keys rewinds the appropriate cartridge.

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

File Directory

In order to save potential wear on a tape, and to permit high-speed directory access, the file directory for a tape cartridge is copied into R/W memory. Whenever access is first made to the cartridge, the information is physically read from the tape. After that, all accesses to the cartridge's files utilize this copy of the directory. The copy is kept updated and current. You will lose (erase) the copy if you –

- Execute RESET ( .
- Execute SCRATCH A.
- Remove the tape from the drive.

The next time the cartridge is accessed after one of these events, the process of copying the directory into memory is repeated.

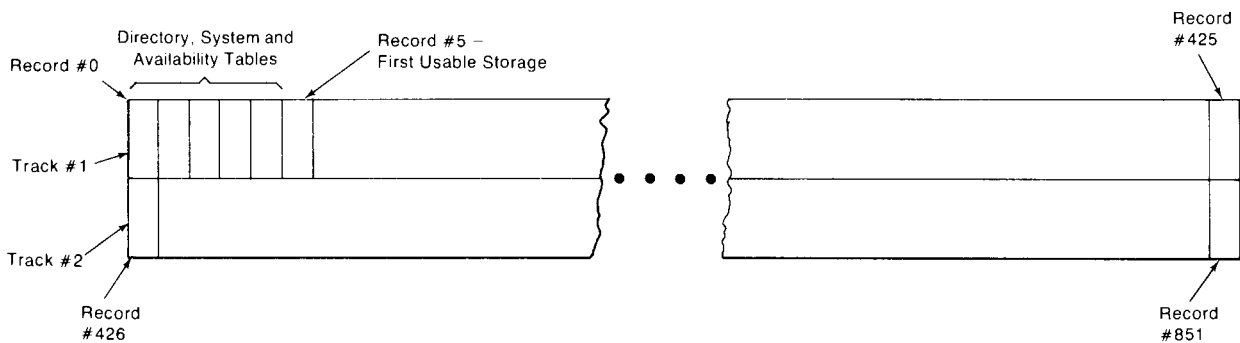
Each time a change in the directory contents occurs (i. e., with a PROTECT, PURGE, RENAME, RE-SAVE, RE-STORE, SAVE, STORE ALL, STORE BIN, or STORE KEY instruction) the contents of the memory version of the directory are written to the tape to assure that the written version accurately reflects the current tape contents.

While the directory is being copied onto the tape, the directory is vulnerable should a RESET (CONTROL STOP) occur. If RESET should be pressed during the copying operation, that operation is aborted and it is probable that the directory will be left in disarray. In such a case it is necessary to re-initialize the cartridge. Similar problems can arise if RESET occurs during other writing operations. To avoid them all, avoid pressing RESET during execution of any mass storage operation.

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. Diagrammatically, 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 –

```

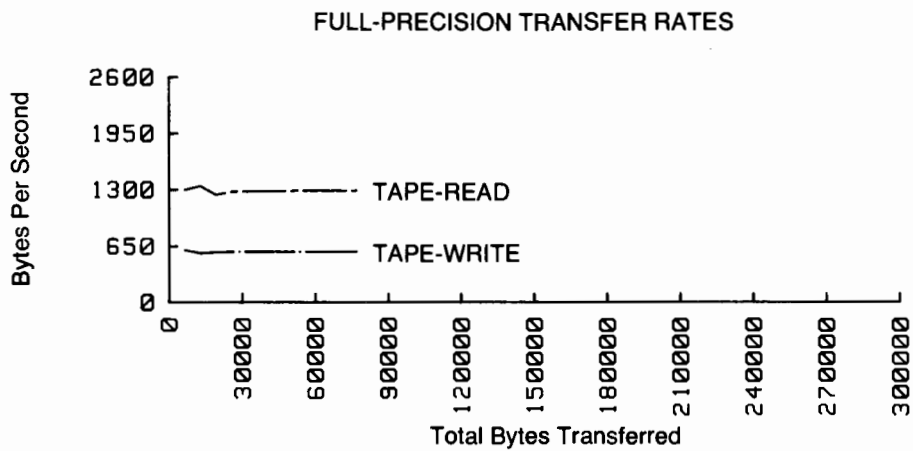
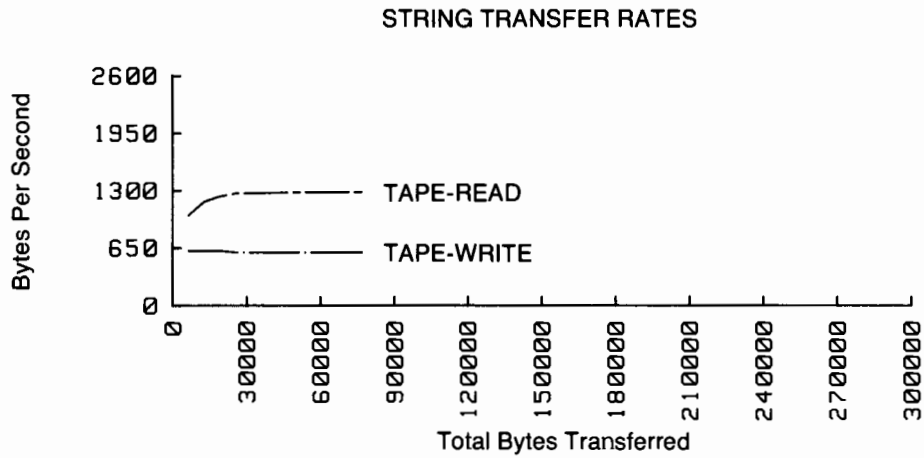
100 INITIALIZE ":T"
110 MASS STORAGE IS ":T"
120 CREATE "DUMMY",420
130 CREATE "+++++",1
140 PURGE "DUMMY"
150 END

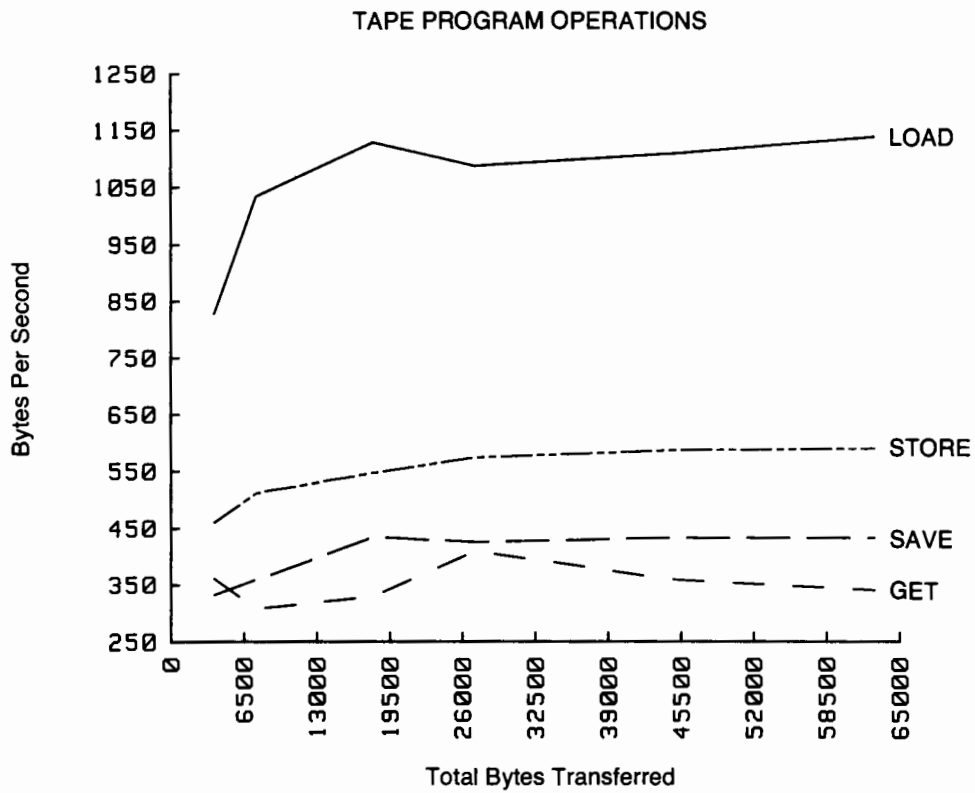
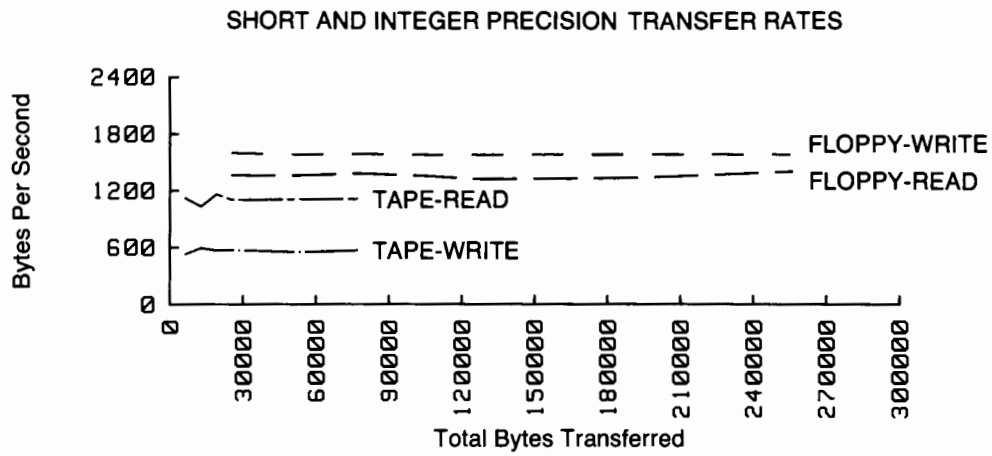
```

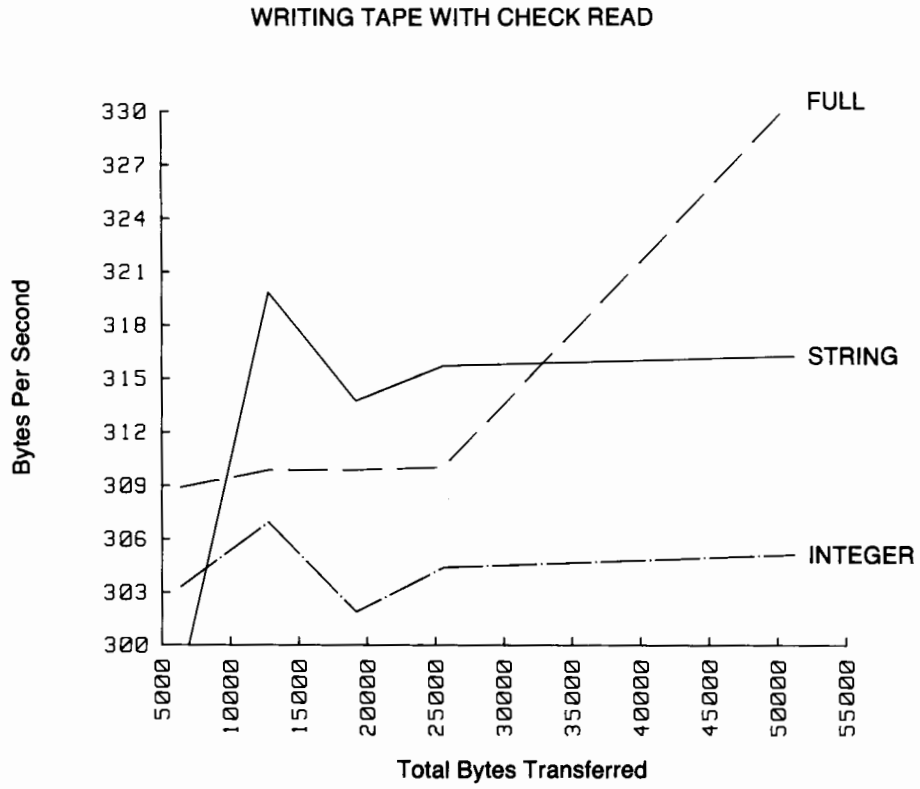
Of course, it is possible to do something similar to this at any time after initialization and before using record 425. After this file is created, however, it is not possible to create files of more than 426 physical records until the file is purged.

Timings

The following graphs present the transfer rates in various situations. They are included as an aid in designing those applications where efficiency or speed is a major consideration.







Notes

Appendix B

Disc Drives

Hewlett-Packard manufactures a number of disk drives, in a wide variety of potential configurations, which may be used with the computer. 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.

Page MS-5 contains a table showing the devices accessible by your Mass Storage ROM. Each drive is supplied with an operating or installation manual to which reference can be made for any information regarding installation, operation, and maintenance of the drive.



Interfacing



Physical connection between your computer and many of the 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 Disk Interface Installation Manual. 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 your desktop computer and the HP 9895A Flexible Disk Drive is accomplished with the HP 98034 HP-IB Interface. Only an HP 7910H, HP 7908, HP 7911, HP 7912, or another HP 9895A may share that HP-IB line. Installation and connection of the drive are discussed in the 9895A Flexible Disk User's Manual (HP part number 09895-90000). The 98034 Installation and Service Manual (HP part number 98034-90000) covers installation of the HP 98034 Interface.

Physical connection between your desktop computer and the HP 7910H Fixed Disk Drive is accomplished with the HP 98034 HP-IB Interface. Only an HP 9895A, HP 7908, HP 7911, HP 7912, or another HP 7910H may share that HP-IB line. Installation and connection of the drive are discussed in the 7910 Disc Drive Installation Manual (HP part number 07910-90902). The 98034 Installation and Service Manual (HP part number 98034-90000) covers installation of the HP 98034 Interface.

Physical connection between your desktop computer and the HP 7908, HP 7911, or HP 7912 Disc Drive is accomplished with the HP 98034 HP-IB interface. Only an HP 7908, HP 7911, HP 7912, HP 7910, or HP 9895A may share that HP-IB line. Installation and connection of the disk is discussed in the disk's installation manual. The 98034 Installation & Service Manual (HP part number 98034-90000) covers installation of the HP 98034 Interface.

Initializing

Executing any mass storage command to an uninitialized (or improperly initialized) hard disk drive or platter will cause your computer to hang. Reset ( ) returns control when this occurs.

Hard disk platters can be initialized individually. To do so, the INITIALIZE statement is used, setting the msus to the location of the platter. This location should always be on a disk drive with a unit number of 0. Any other unit number will cause incomplete initialization of the platter.

If there is more than one hard disk connected to your computer, all attempts to initialize a drive or a platter must be done on units with the drive numbers set to 0. To ensure that no data is inadvertently lost, drives should be set to 0, 1, 3, or 7 when disk initializations are made.

Timings

When designing a program, you are often concerned with the amount of time needed to exchange programs/data between the computer and an external mass storage device. If this was a function of only one factor, such as the BASIC statement initiating the transfer, it would be a simple matter to choose the statement that would yield the transfer rate needed for your application. Unfortunately, the transfer rate of data and programs between a mass storage device and your desktop computer is a complex intermingling of many different factors.

For example, the transfer rate between an HP7925 disc drive and an HP9845 B/C desktop computer is higher than that of the HP7908 disc drive for most BASIC statements. However, when using the MAT READ# statement, the transfer rate for the HP7908 disc drive is nearly equal to that of the HP7925.

Some of the more important factors affecting the transfer rate are listed below:

- Mass Storage Hardware- The transfer rate is dependant upon the mass storage device used (such as the HP9895A flexible disc drive versus the HP9885M flexible disc drive).
- Computer Hardware- The transfer rates also depend upon the computer hardware used. With the 9845 desktop computer, the transfer rate depends upon whether the computer is equipped with the standard language processor or the faster bit slice language processor. Transfer rates for the faster bit slice processor should be equal or faster in almost all cases. Data taken for the graphs that follow was obtained from a 9845 with the standard language processor.

Transfer rates for the HP 9835 desktop computer are nearly identical to those of the HP 9845 with the standard language processor. Note that the mass storage devices supported by the 9845 desktop computer are not necessarily the same as those supported by the 9835 desktop computer.

To help you design your application programs, we have provided several graphs demonstrating the effect of the more important factors. Since it is extremely difficult to represent the relationship between all factors affecting the transfer rate, we have chosen to show the effect of varying one factor while the other factors are held constant.

All data for the graphs was taken with a 9845B desktop computer equipped with the standard language processor. Most of the transfer times shown in the graph were acquired using a single BASIC statement to transfer the data. For example, in Graph 1 the transfer rate for transferring 100k bytes of data with the FREAD statement from an HP7925 disc to the 9845 was the result of a single FREAD statement. When the transfer times/rates represented on the graph are not the result of a single statement transfer, partial program listings are provided with the graphs.

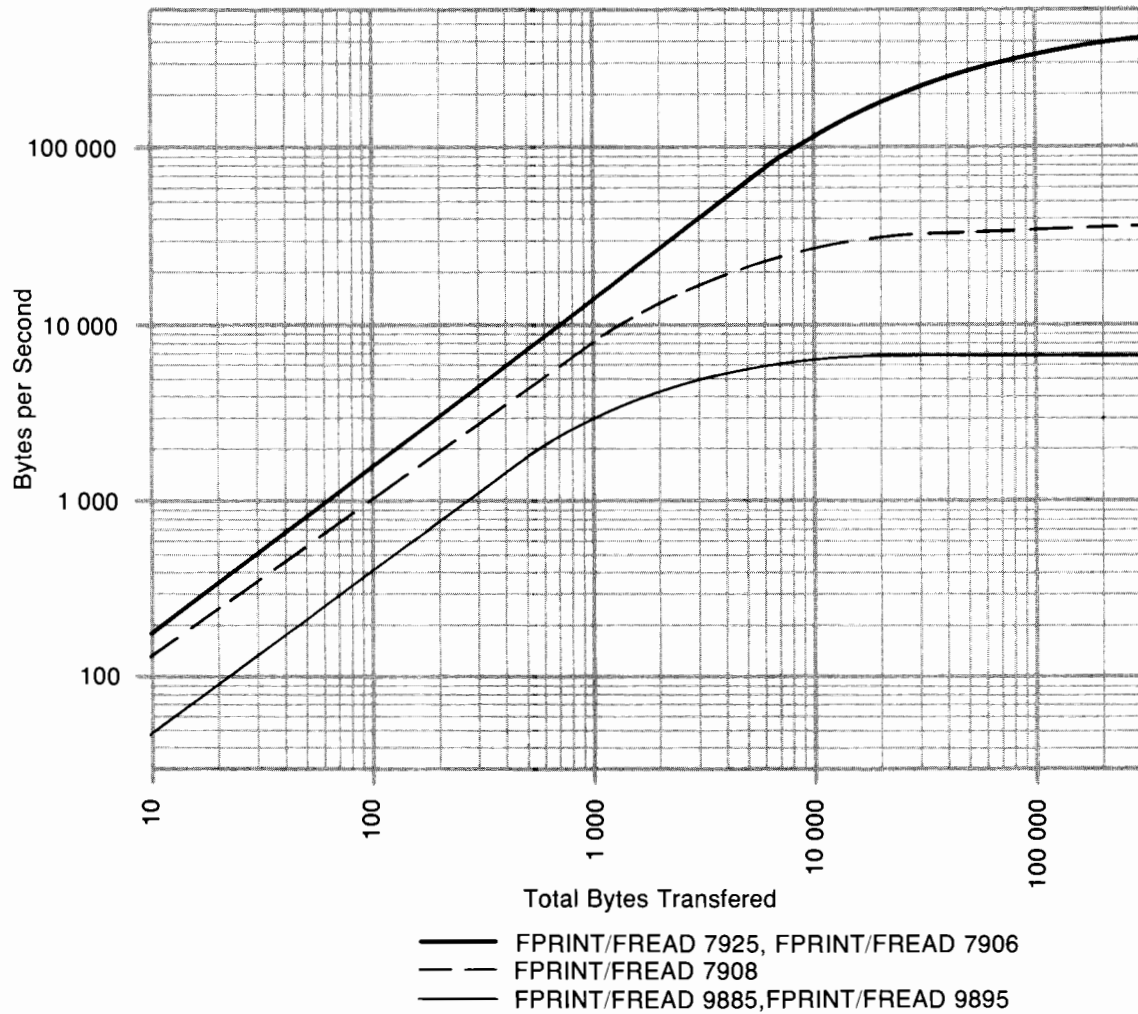
The title for each graph indicates the computer and mass storage hardware used, the type of data that was transferred (such as string data, real data, etc), the number of bytes per defined record on the mass storage device, the BASIC statement initiating the transfer, and the number of files in the directory.

The first group of graphs (graphs 1, 2, and 3) show how the transfer rates is effected by the mass storage device used in the transfer. Data is transferred between an HP9845 B/C desktop computer equipped with the standard language processor and a variety of mass storage devices (HP7906, HP7908, HP7925, HP9885M, and HP9895A). All data was acquired using single statement transfers.

Several different mass storage commands were used to produce the graphs, thus showing that individual devices provide faster or slower transfer rates, depending on the BASIC language statement initiating the transfer. For example, in graph 1 you can see that the transfer rate for the HP7925 disc is higher than that of the HP7908 when the transfer has been initiated with the FREAD or FPRINT statement. However, in graph 3 you can see that when the transfer is initiated by the MAT READ# statement, the transfer rates of the two devices are almost identical.

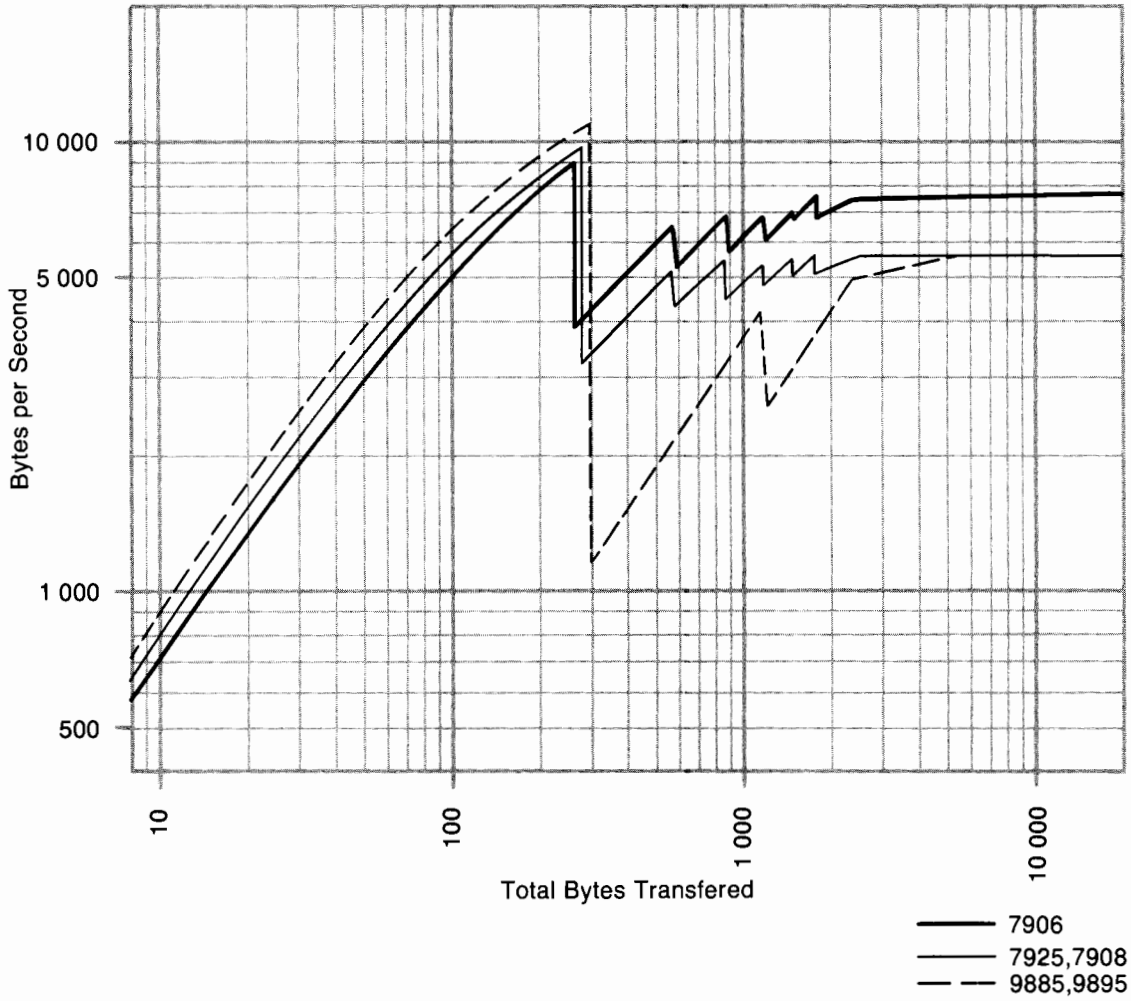
Notice that graphs 2 and 3 show definite peaks in the transfer rate. These peaks occur at 256 total bytes transferred and integer multiples of 256 total bytes transferred.

MASS STORAGE DEVICES
 9845B/C-7906,7908,7925,9885,9895
 Single File in Directory, FPRINT/FREAD String, 256 Bytes/Rec



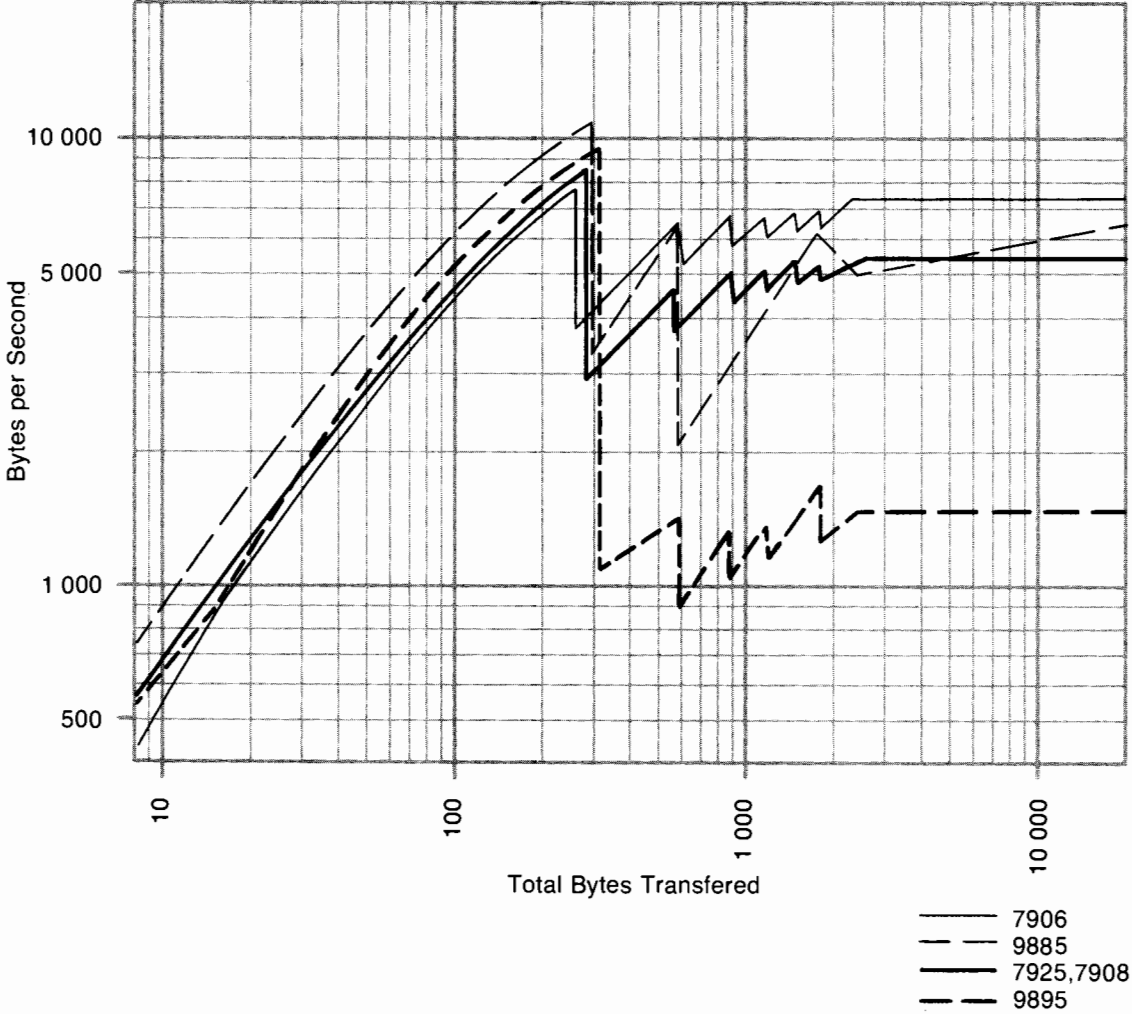
Graph 1

MASS STORAGE DEVICES
9845B/C-7906,7908,7925,9885,9895
Single File in Directory, MAT PRINT# Real, 256 Bytes/Rec



Graph 2

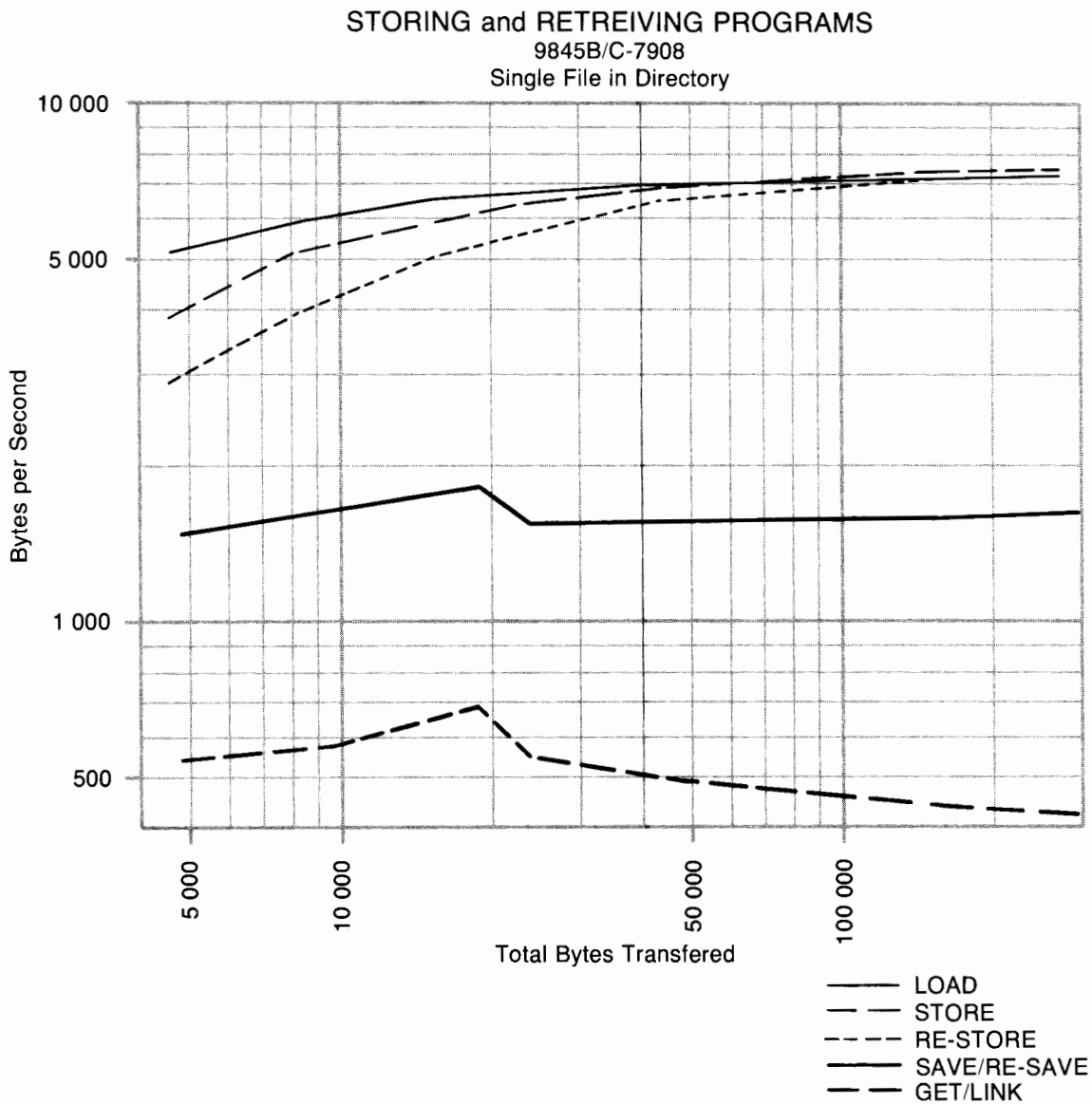
MASS STORAGE DEVICES
9845B/C-7906,7908,7925,9885,9895
Single File in Directory, MAT READ#, REAL, 256 Bytes/Rec



Graph 3



Graph 4 shows the transfer rate between an HP9845B/C desktop computer equipped with the standard language processor and an HP7908 disc drive for transferring programs. Mass storage statements showing the differences between the two methods of placing programs on a mass storage device (STORE vs SAVE) and the methods of retrieving programs from a mass storage device (LOAD vs GET) were used to create the graph. The data for this graph was generated with only one file in the directory of the 7908. As the number of files in the directory increases, the time required to search for the file increases, thus increasing the apparent transfer rate.



Graph 4

Graph 5 shows the differences in the transfer rate caused by the different BASIC language statements used to transfer data between a desktop computer and a mass storage device. In this case an HP9845B/C desktop computer equipped with the standard language processor and an HP7908 disc drive are used to show the differences between using: FREAD/FPRINT, MAT READ#/MAT PRINT#, READ#/PRINT#, and MAT READ#/MAT PRINT# (with CHECK READ).

Note that the type of data transferred is real data for all BASIC statements except FPRINT and FREAD. For these statements, a 12 character string (16 bytes) was used to measure the transfer rate. With these statements, little difference in the transfer rate is expected for the two data types.

The data for this graph was taken using a disc that had a single file in the directory. The file written to and read from had 256 bytes/record. Again, the graph shows that mass storage statements utilizing the computer's internal buffer have marked peaks in their transfer rates at integer multiples of 256 total bytes transferred.

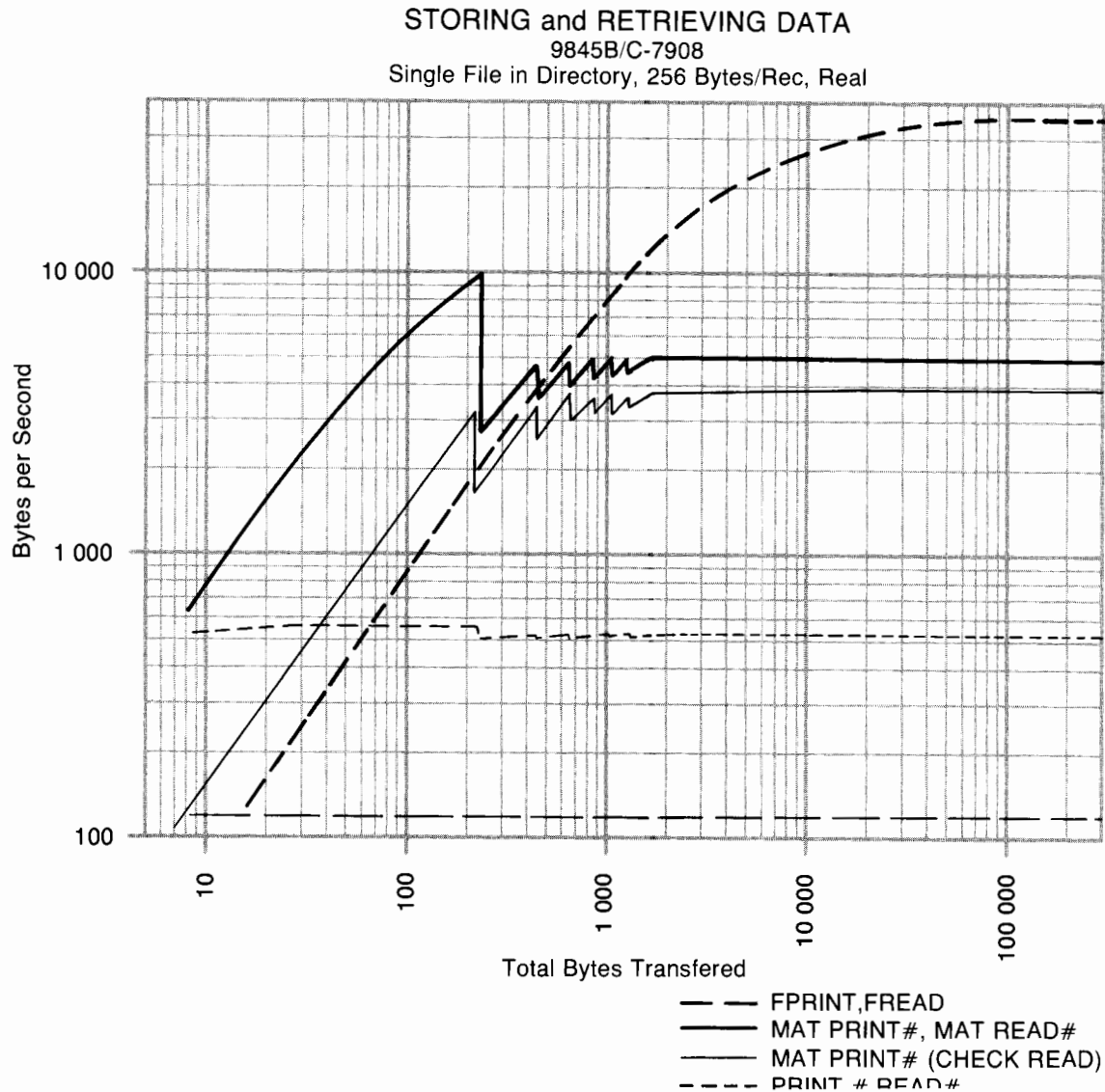
The transfer rate curves for PRINT # and READ # were obtained with the following group of basic statements:

```

50 FOR I= 1 to Length  !Where Length is the total number of bytes
60                    !transferred divided by B (since each real
70                    !value requires B bytes of storage).
80  PRINT #1;R         !Where R is a real variable.
90 NEXT I

```

The same block of statements applies to the transfer rate curves developed for the READ # statement. As you can see, a transfer that is not performed with a single statement, such as FREAD or MAT READ #, requires the execution of other BASIC statements. The execution of these statements decreases the transfer rate since it takes time for these extra statements to execute each time through the loop.

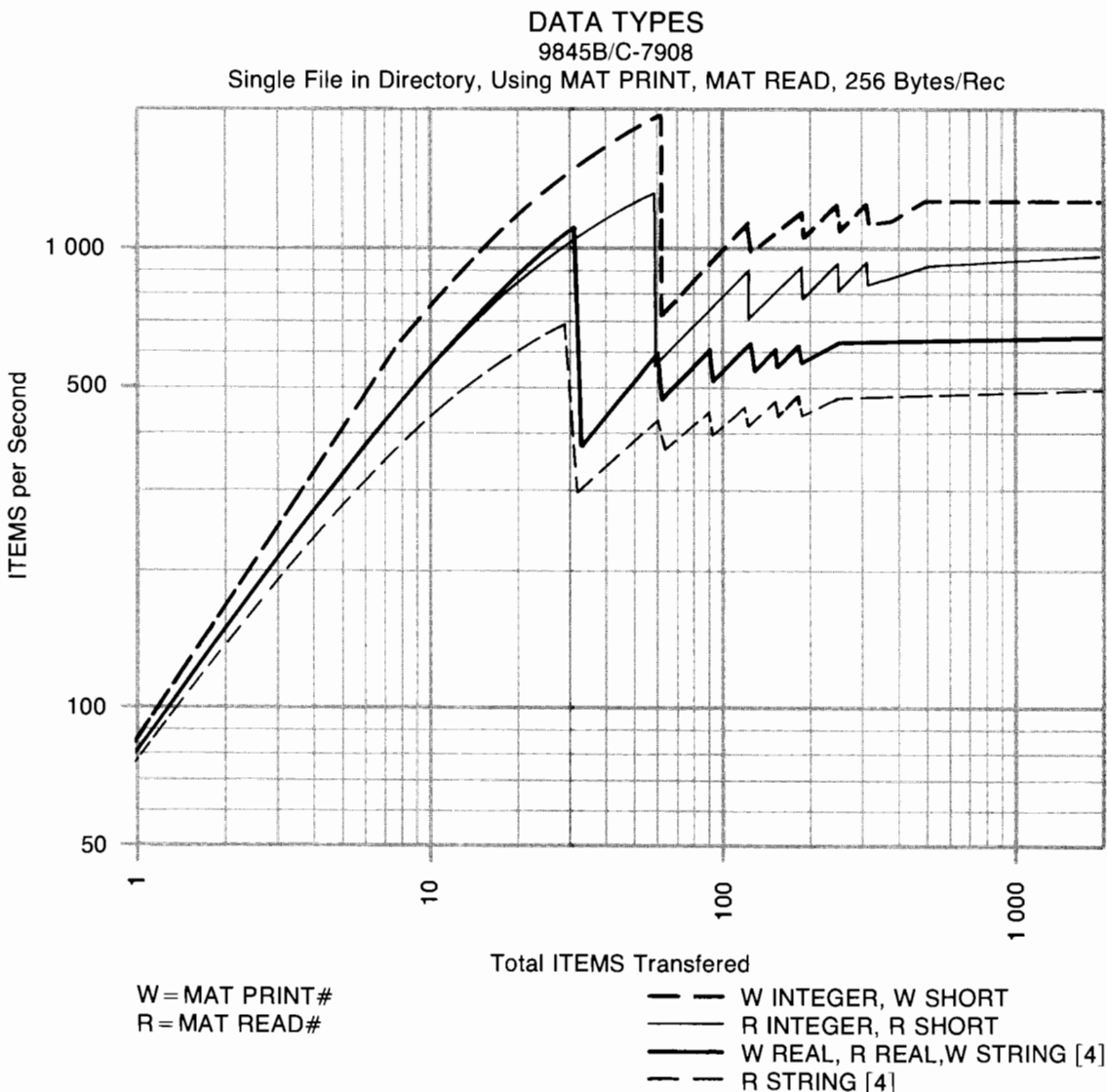


Graph 5

More often than not, we are concerned with the number of data items that can be transferred, and not the number of bytes that can be transferred. For example, in an application where the desktop computer is being used to take 300 readings per second from a particular test device, the programmer is concerned with the quickest method to place the 300 data items on the mass storage device. If he were concerned with only the fastest transfer rate, looking at graph 5 he would choose to represent the data as string data and use the FPRINT and FREAD statements to transfer his data to and from the disc. However, if the number of bytes per data type is taken into affect, an entirely different approach is suggested.

Graph 6 shows the affect of considering the number of items (where an item is a real number, an integer, a short precision real number, or a string of characters) transferred instead of the number of bytes transferred. From this graph it can be seen that for the MAT PRINT # and MAT READ # statements, the highest number of data items transferred per second occurs if the data that is being written or read is either integer data or short precision real data. The string data type transfers shown in the graph represent transfers of 4 character strings (8 bytes of disc storage).

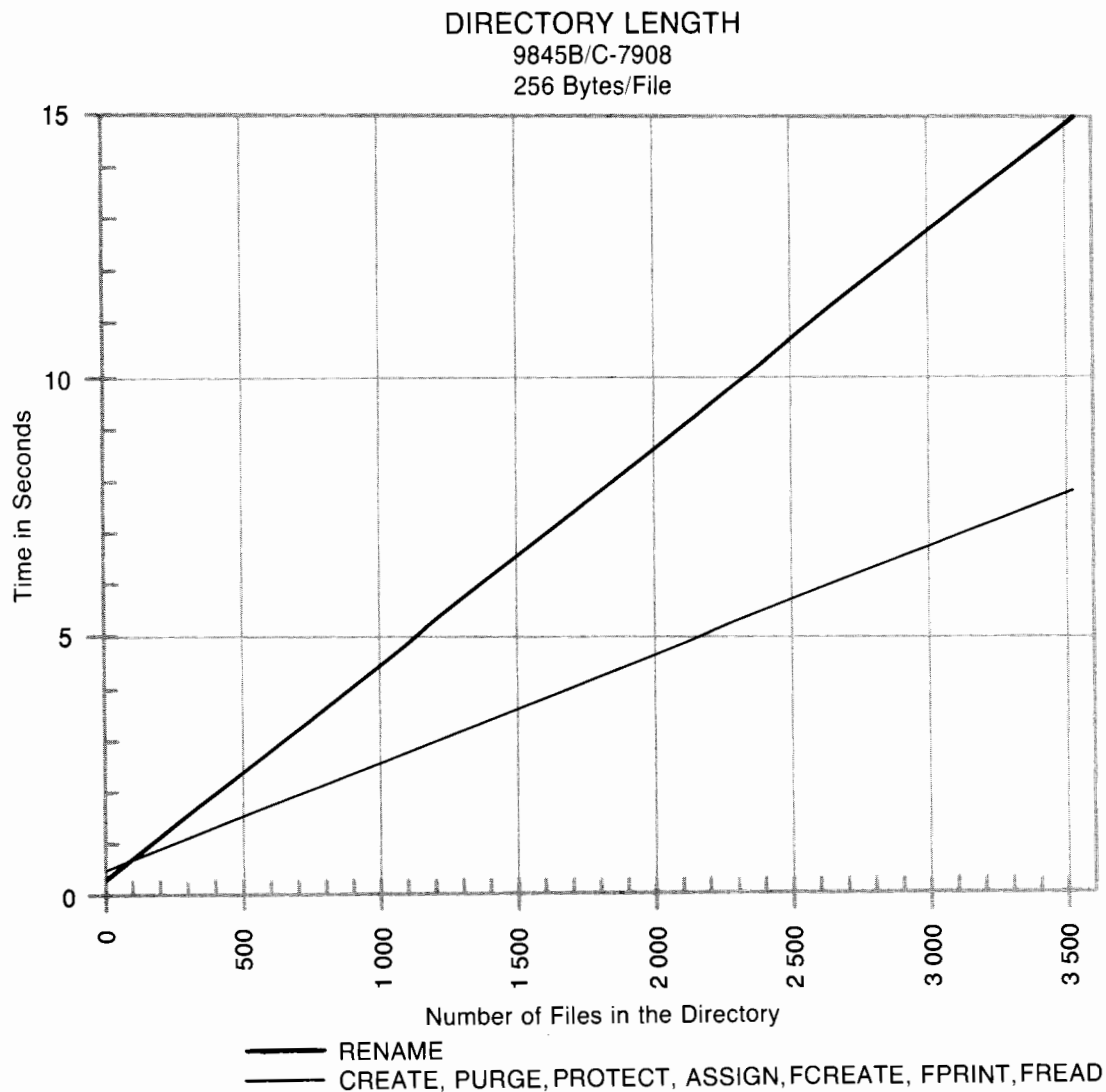
The data for this graph was generated using the HP9845B/C desktop computer with standard language processor and the HP7908 disc drive. The 7908 had a single file in the directory with 256 bytes per record. The BASIC language mass storage statements, MAT READ and MAT PRINT, were used to generate the data for the graph.



Graph 6

Graph 7 shows you the effect of the number of files in the directory on the time required to access the last file in that directory. The data for this graph was generated using an HP9845B/C desktop computer with standard language processor and an HP7908 disc drive. The graph shows the time required to access the 7908 disc drive with a particular BASIC language mass storage statement as the number of files in the directory increases. Notice that the amount of time required for the RENAME function is exactly double that of the other types of access. This is because two disc accesses are required for the RENAME statement, as opposed to one disc access for each of the other commands represented.

The graph shows that fewer files per disc requires less amount of search time to find the file, thus increasing the apparent transfer rate.

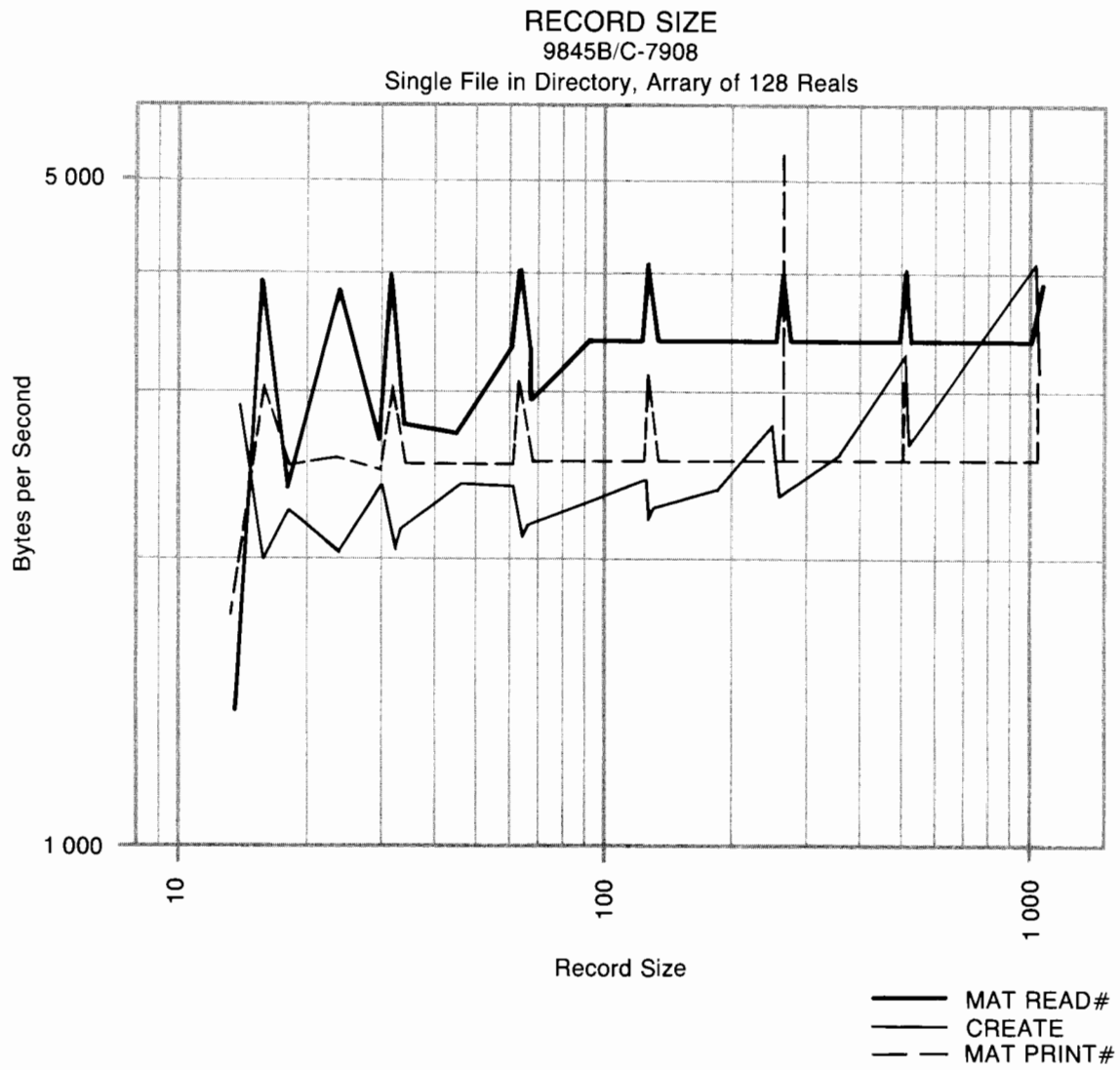


The next two graphs show the effect of record size on the transfer rate for hard disc drives (represented by the HP7908 disc drive in graph 8) and flexible disc drives (represented by the HP9895A disc drive in graph 9). Graph 8 shows the effect of the record size when creating a file on a 7908 disc drive with an empty directory and then writing to and reading from that file with the MAT PRINT# and MAT READ# statements. An array of 128 reals is used with the MAT PRINT# and MAT READ# statements.

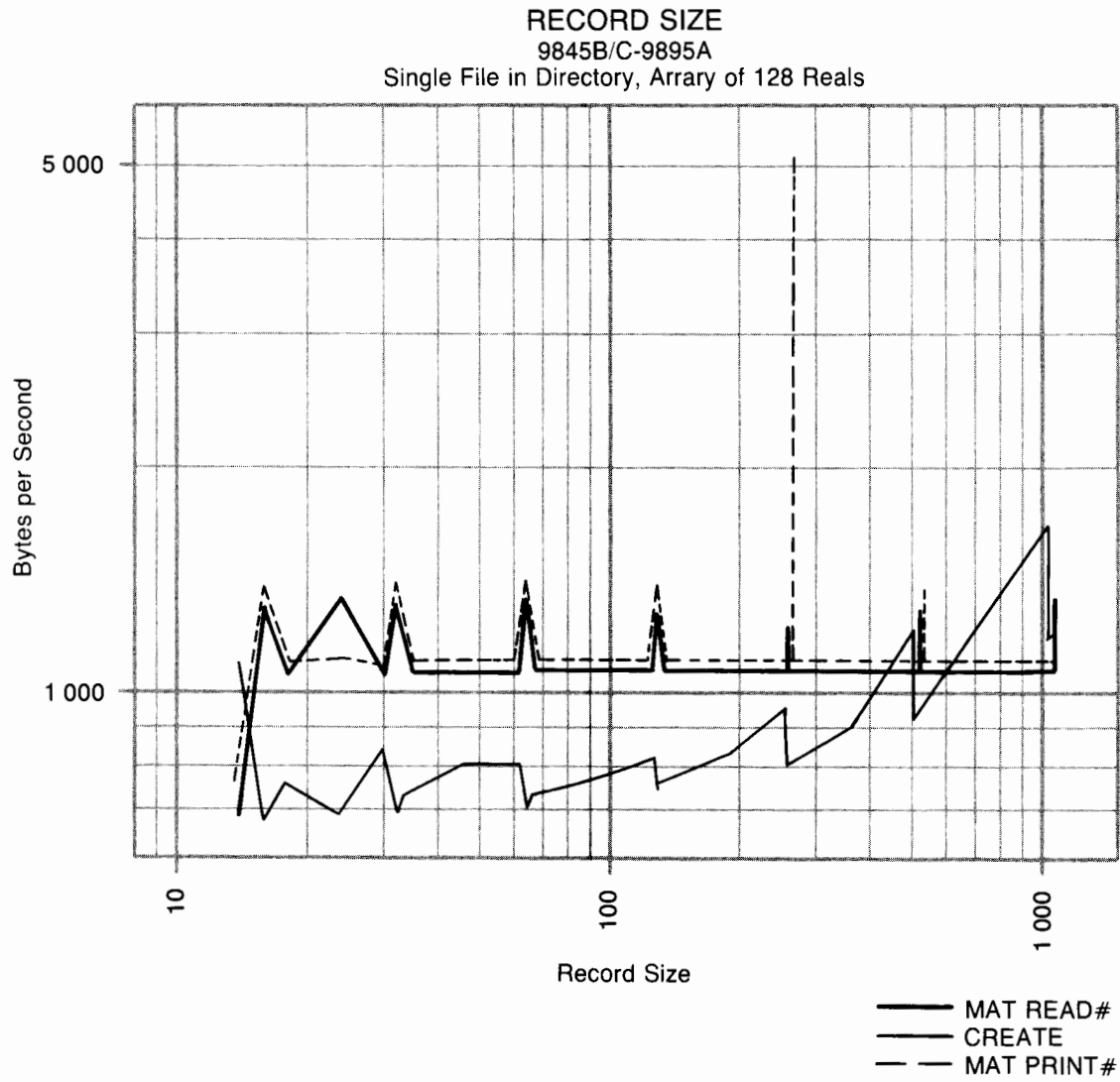
Graph 9 shows the effect of the record size when creating a file on a 9895A flexible disc drive with an empty directory and then writing to and reading from that file with the MAT PRINT# and MAT READ# statements. An array of 128 reals is used with the MAT PRINT# and MAT READ# statements.

Notice in both graphs that peaks are generated at multiples of (and factors of) 256 bytes. This is due to the 9845's internal 256 byte buffer and the number of bytes per defined record. When the size of each record is a factor of 256 (such as 16, 32, 64, and 128), transferring the internal buffer results in completely filled records. For example, if the record size of a file is 64, then transferring the buffer exactly fills 4 records. Extra time is required for transfers that do not exactly fill a record. A new file always begins at the beginning of a new physical record.

Additionally, you can see that on each graph a large peak occurs for transfers to 256 byte records with the MAT PRINT # statement. This is caused by the fact that for transfers other than 256 bytes, the system performs a read before each write with the MAT PRINT # (and PRINT #) statements. Only for transfers of 256 bytes does the system not perform a read before a write.



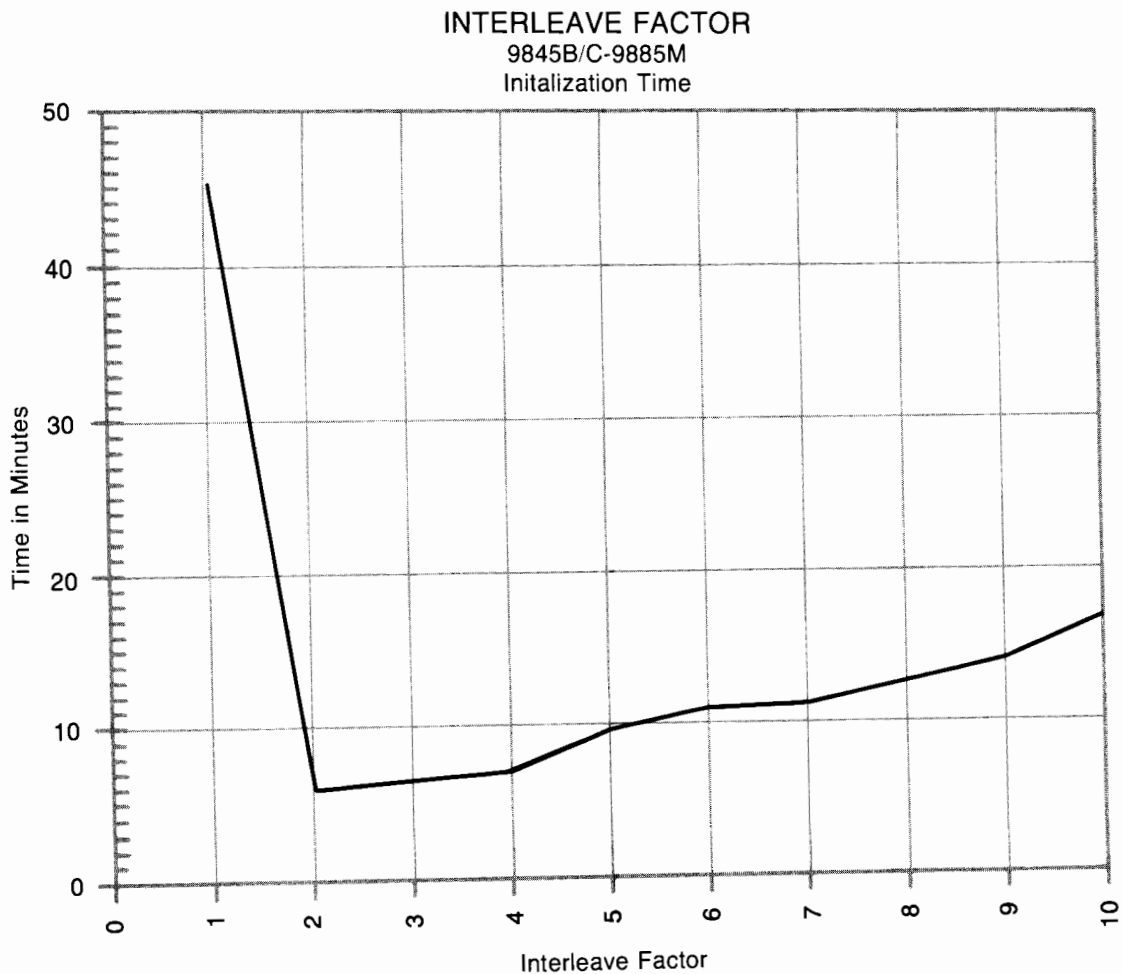
Graph 8



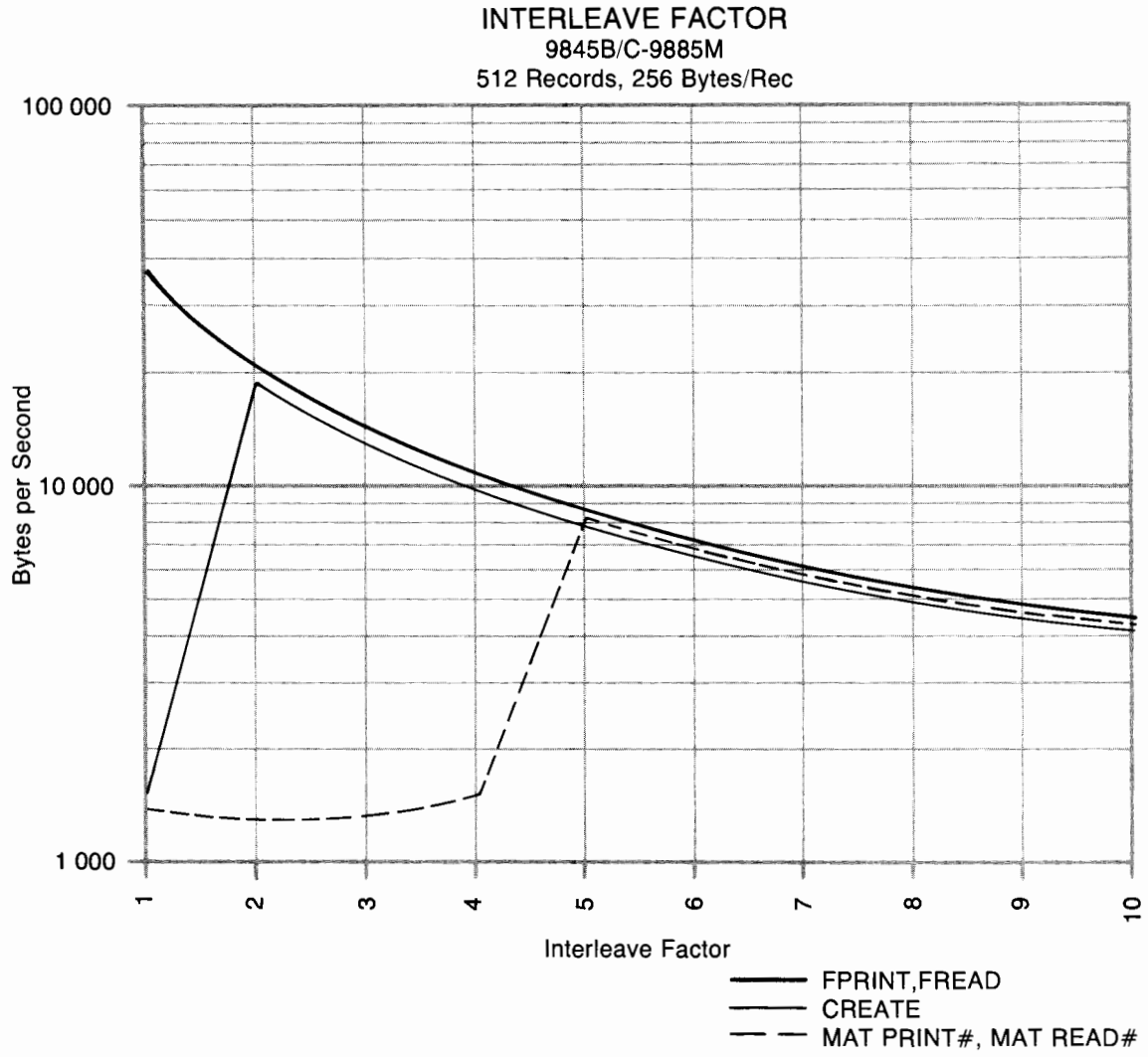
Graph 9

Graph 10 shows the effect on initialization time that the interleave factor has when initializing flexible discs. Graph 11 shows the effect of the interleave factor on the transfer rate when accessing flexible discs. These graphs were generated with the HP9845B/C desktop computer with standard language processor and the HP9885M flexible disc drive. Graphs 12 and 13 show the identical information for the HP9895A flexible disc drive.

Notice that while graph 10 shows that the quickest initialization of a 9885 flexible disc occurs when using the interleave factor 2, graph 11 shows that interleave factor producing the highest transfer rate depends on the BASIC language statement used to access the disc. Taking all mass storage accesses into account, the interleave factor 7 (the default interleave factor when initializing a 9885 disc) provides the best overall access time.



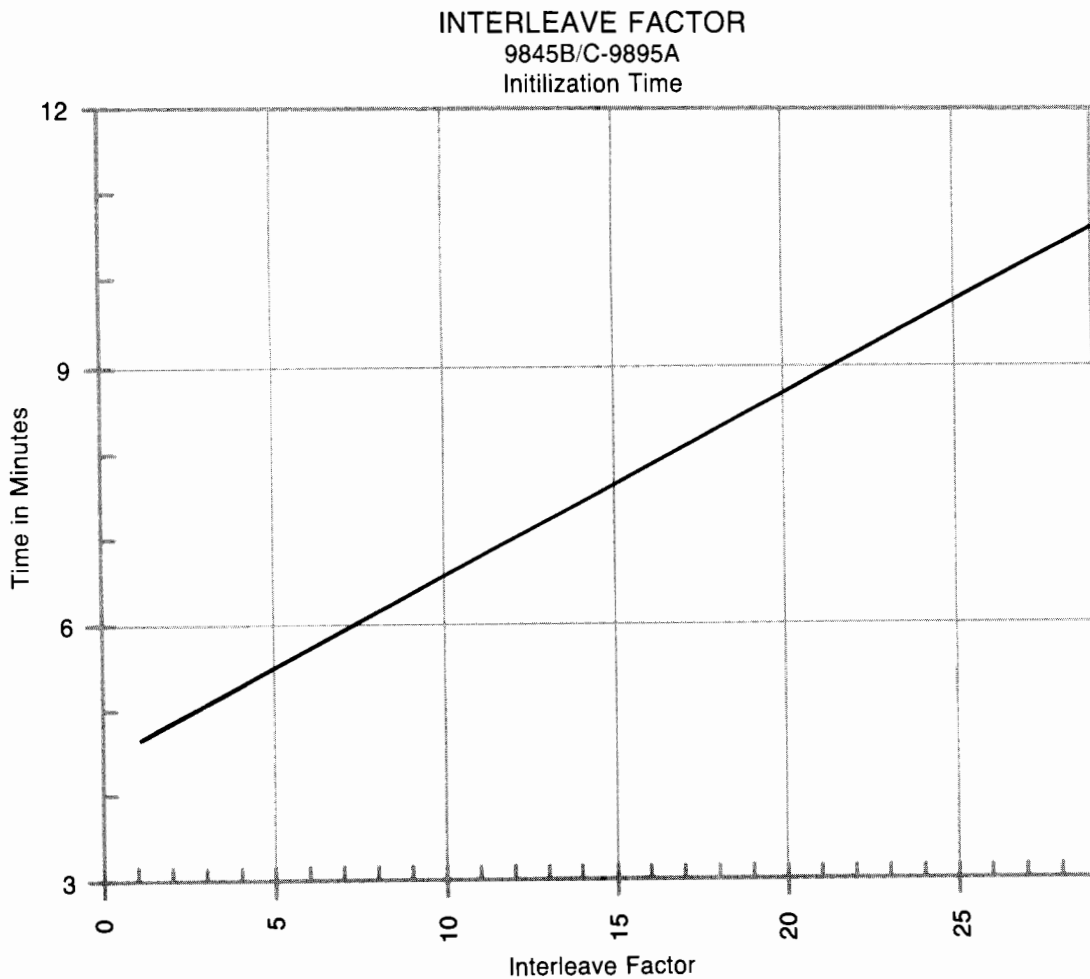
Graph 10



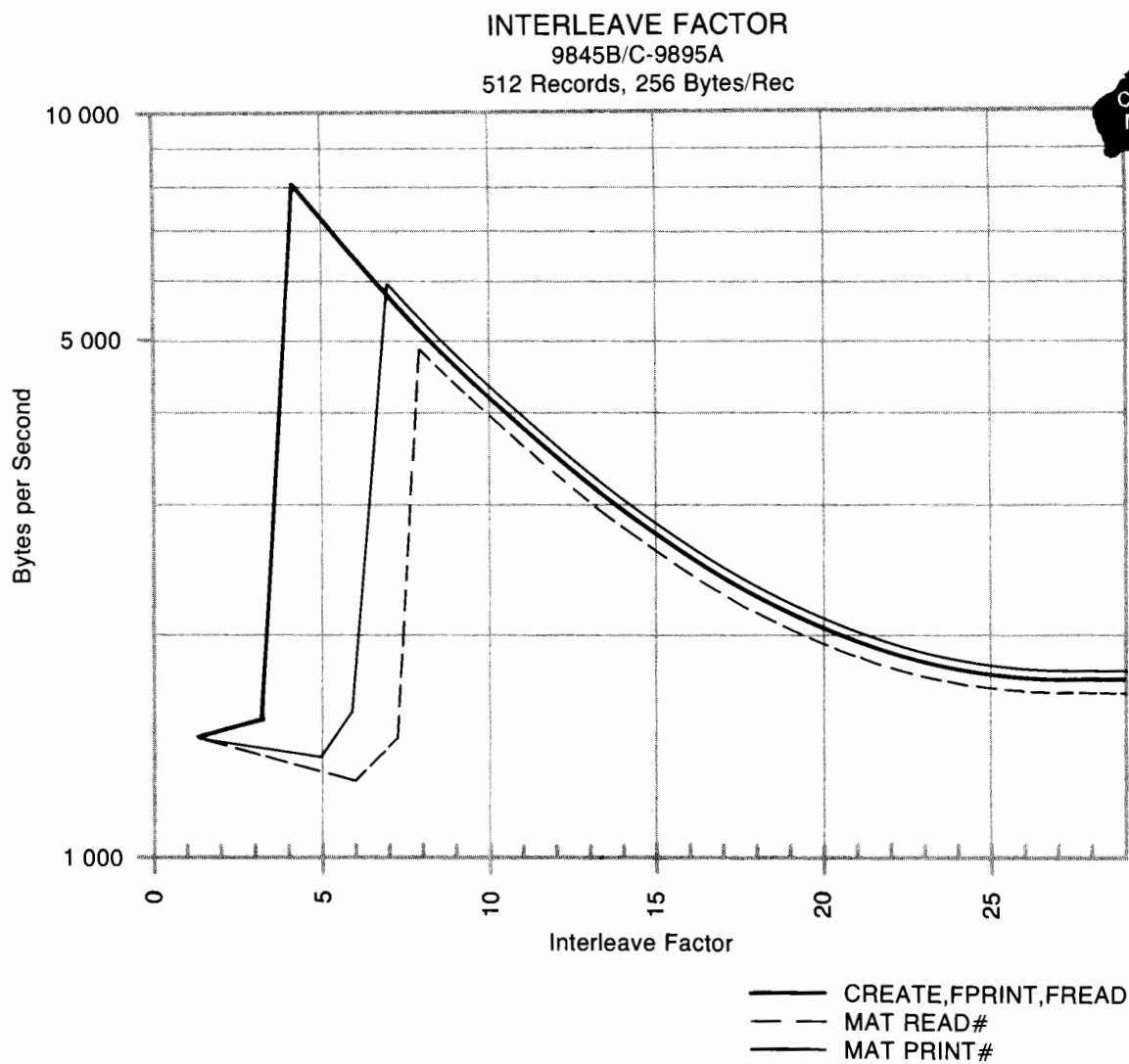
Graph 11

Graph 12 shows the effect on initialization time that the interleave factor has when initializing flexible discs. Graph 13 shows the effect of the interleave factor on the transfer rate when accessing flexible discs. These graphs were generated with the HP9845B/C desktop computer with standard language processor and the HP9895A flexible disc drive. Graphs 10 and 11 show the identical information for the HP9885M flexible disc drive.

Notice graph 12 shows that the quickest initialization of a 9895 flexible disc occurs when using the interleave factor 1. However, graph 13 shows that the interleave factor producing the highest transfer rate depends on the BASIC language statement used to access the disc. Taking all mass storage accesses into account, the interleave factor 11 provides the best overall access time for the 9895 flexible disc drive.



Graph 12



Graph 13

Graph 14 shows the difference between random access and serial access to an external mass storage device. The data for this graph is taken from an HP9845B/C desktop computer with standard language processor and an HP7908 disc drive. A single file was present in the directory of the 7908, and that file was created with 256 bytes per record. A string data type is transferred serially or randomly between the disc and the desktop. The size of this string is varied to show the effect of differing numbers of bytes being transferred.

The difference between random and serial transfers can be attributed to the way that the desktop handles each type of transfer. A serial transfer first fills the desktop's internal 256 byte buffer and then performs a physical transfer to/from the disc. However, a random transfer performs a physical transfer each time the random transfer statement occurs, regardless of whether the desktop's internal 256 byte buffer is filled or not.

For the two types of transfers, the following block of statements was used to generate the graphs:

Random transfer

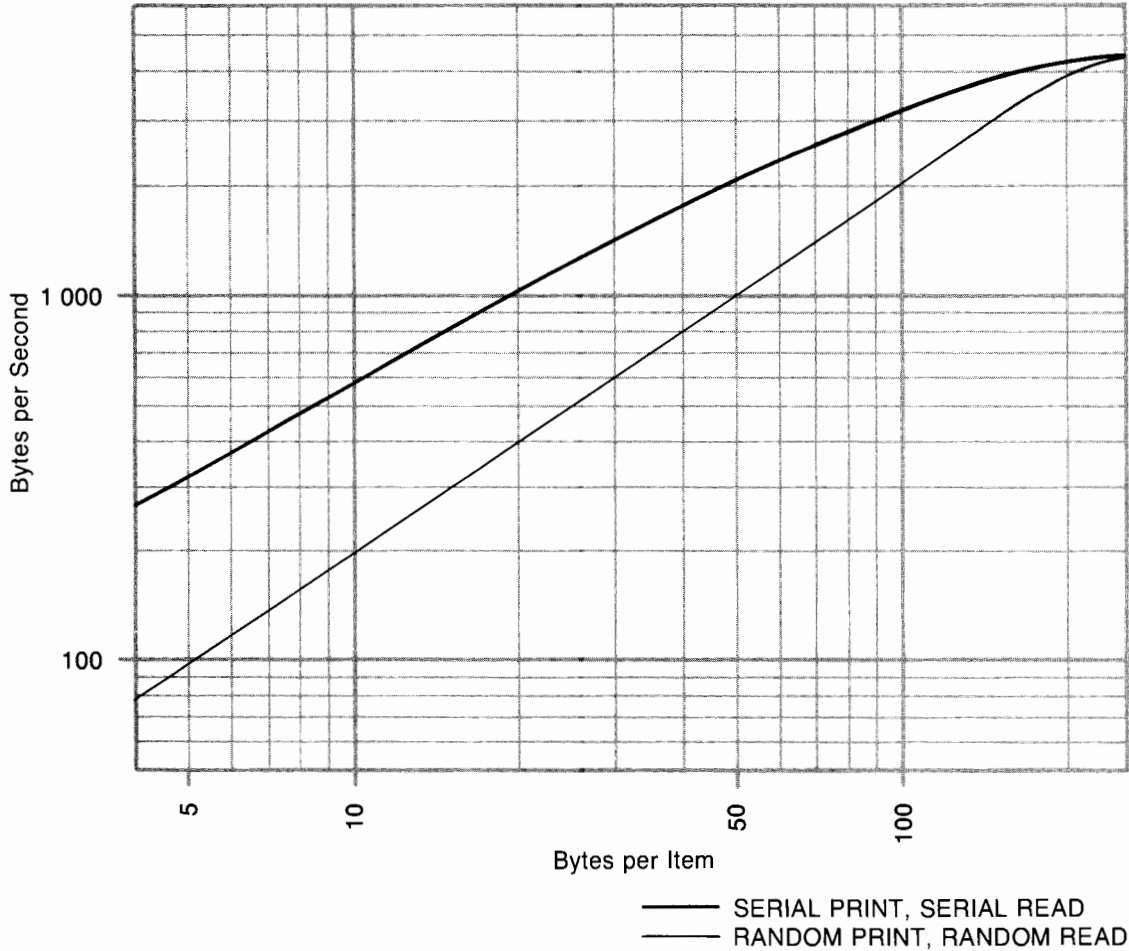
```
50 FOR I = 1 TO 32
60   PRINT#1, I; A$
70 NEXT I
```

Serial transfer

```
50 FOR I = 1 TO 32
60   PRINT#1; A$
70 NEXT I
```

By changing the size of the string variable A\$, data was obtained for strings variables of size 0 bytes (null string) to 256 bytes.

RANDOM vs SERIAL
9845B/C-7908
Single File in Directory, 256 Bytes/Rec



Graph 14

Notes



Rating Characteristics

7912	7920	7925	9885	9895A Dual Drive Master or Opt 012 Dual Drive Slave	9895A Opt 010 Single Drive Master or Opt 011 Single Drive Slave	7910H
hard	Hard	Hard	Flexible	Flexible	Flexible	Hard
4 978 944	48 758 784	117 014 528	499 200	2 273 280	1 136 640	11 943 936
258 824	190 464	457 088	1 950	8 880	4 440	46 656
3 962	3 968	7 142	65	296	148	1 458
12 944	8 000	19 024	352	704	352	2 288
256	256	256	256	256	256	256
64	48	64	30	30	30	32
582	800	800	67	77	77	735
3 600 cylinder 7	3 600 rpm Cylinder 5	2 700 rpm Cylinder 9	360 rpm -- 1	360 rpm Cylinder 4	360 rpm Cylinder 2	3 000 rpm Cylinder 2
35 ms	25 ms*	25ms	267 ms	77 ms	77 ms	70 ms
8.3 ms	8.3 ms	11.1 ms	83 ms	83 ms	83 ms	10 ms
--	--	--	8 ms	20 ms	20 ms	--
5 ms	5 ms*	5 ms	8 ms*	3 ms	3 ms	10 ms

Disc Drive Op

	7905A	7906	7908	7911
Type of Disk	Hard	Hard	Hard	hard
User Storage Capacity:				
Bytes	4 866 048 Fixed 9 732 096 Removable	9 732 096 Fixed 9 732 096 Removable	16 432 640	27 820 032
Physical records	19 008 Fixed 38 016 Removable	38 016 Fixed 38 016 Removable	64 190	108 672
Tracks	396 Fixed 792 Removable	792 Fixed 792 Removable	1 834	1 698
Files (maximum)	1 136 Fixed 2 288 Removable	2 288 Fixed 2 288 Removable	3 552	6 112
Device Capacity:				
Bytes per record	256	256	256	256
Records per track	48	48	35	64
Tracks per surface	400	800 Fixed 400 Removable	370	582
Accessing:				
Rate of spin	3 600 rpm	3 600 rpm	3 600 rpm	3 600
Access mode	Surface	Surface	Cylinder	cylinder
Number of heads (surfaces)	1 Fixed 2 Removable	1 Fixed 2 Removable	5	3
Average seek time	25 ms*	25 ms*	42 ms	35 ms
Average rotational delay	8.3 ms	8.3 ms	8.3 ms	8.3 ms
Head settling time	--	--	--	--
Step time	5 ms*	5 ms*	5 ms	5 ms

* Includes head settling time.

