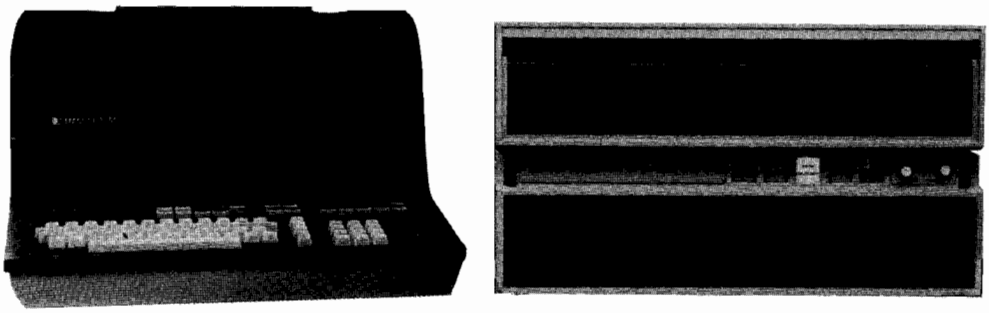
HEWLETT-PACKARD 9830A CALCULATOR
9880A/B MASS MEMORY

OPERATING MANUAL

OPERATING MANUAL

HEWLETT-PACKARD 9880A/B MASS MEMORY



9880A/B MASS MEMORY

HEWLETT–PACKARD CALCULATOR PRODUCTS DIVISION
P.O. Box 301, Loveland, Colorado 80537, Tel. (303) 667-5000
(For World-wide Sales and Service Offices see rear of manual.)

PREFACE

The desire for speed and accuracy in calculations prompted scientists and engineers to develop high-speed, sophisticated programmable calculators, like the -hp- Model 9830A. The problem of limited storage space, however, has always been frustrating. A calculator with the capability of manipulating large amounts of information, accurately and quickly, when there is no equally fast and accurate method of permanently storing this information, is analogous to a powerful navy in a land-locked country.

Your mass memory system represents a revolutionary solution to this problem in the programmable calculator industry. Use of the mass memory system helps fulfill the Model 30's potential, providing you with the kind of storage unavailable in any self-contained calculator today.

A vast amount of information can be stored and accessed on each platter used with the mass memory system. In fact, the information contained on one platter would require a tape cassette over 1¾ miles long or a stack of data processing cards over 17 feet high! More than three hundred thousand 12-digit items can fit onto one platter. Combine this storage capacity with access time on the order of *milli*seconds, and you can begin to appreciate the power at your fingertips with the mass memory system.

This book is organized so that it can be used in either of two ways:

- A logical arrangement of topics lets you read it straight through, if desired.
- Major topics are self-contained, wherever possible, so it is not necessary to read an entire chapter to extract one idea.

Your mass memory system can be used easily for storing many entire programs. The system is most powerful, however, when used to store and access large quantities of data. Over half of this manual describes data accessing procedures. An understanding of the structure of mass memory files is indispensable for efficient use of the files you create. For this reason, Appendix A should be read over briefly to acquaint you with data file structure before you read Chapter 3, which describes particular methods of storing and retrieving data.

TABLE OF CONTENTS

CHAPTERS

CHAPTERS

APPENDICES

TABLES

FIGURES

Chapter 1
GENERAL INFORMATION

INTRODUCTORY DESCRIPTION

The -hp- 9880A/B Mass Memory System offers the advantages of a very large tape cassette. The amount of available storage space is 2.4 million data bytes (1.2 million words) per platter. In addition, short data access time makes the system extremely functional and easy to use. The average access time is less than 50 milliseconds. With optional applications pacs, the mass memory system is even more powerful; data and program control is increased.

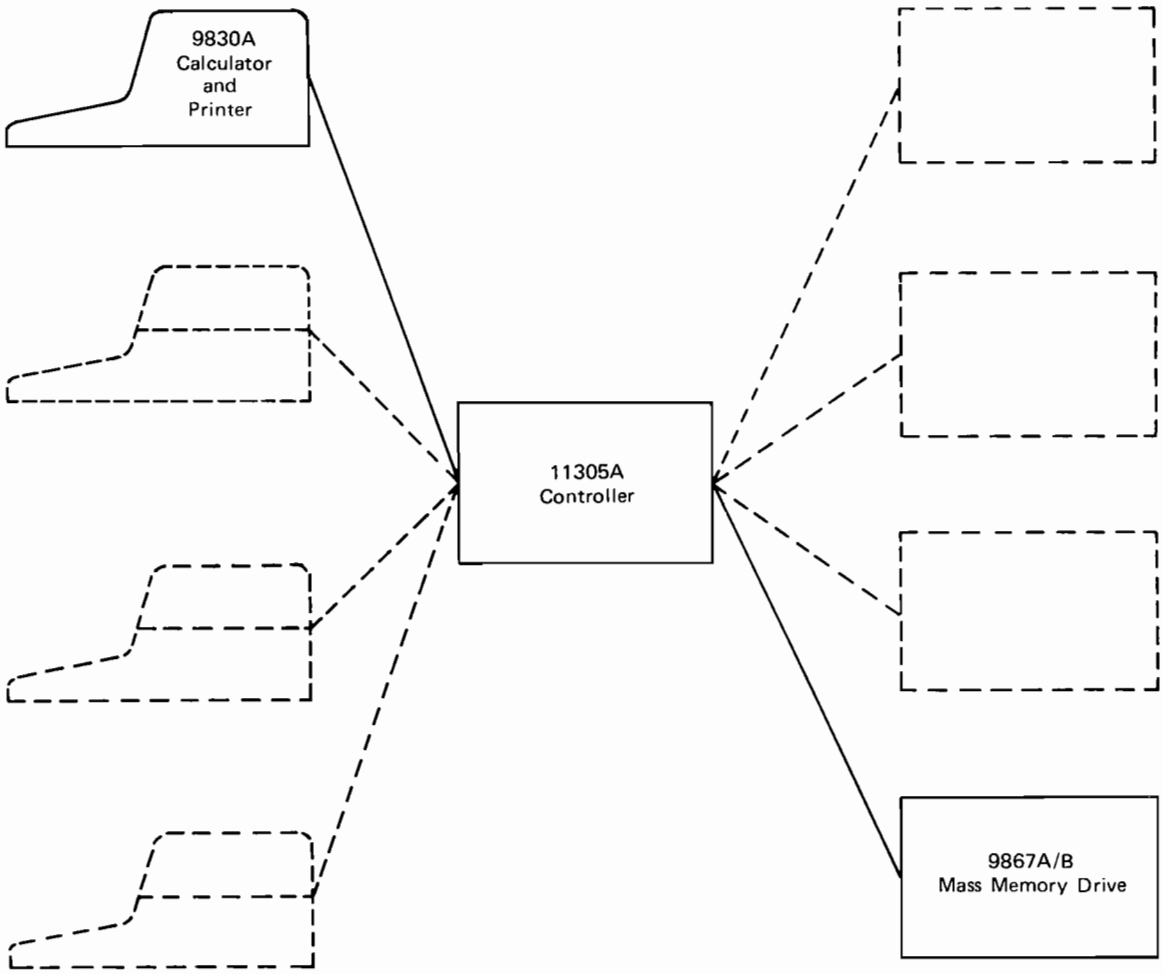


Figure 1-1. Possible Configurations

The mass memory system can consist of a variable number of components, depending on your needs. Possible configurations are shown above (Figure 1-1). Each component is discussed briefly below.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

INTRODUCTORY DESCRIPTION

(Continued)

9830A CALCULATOR

The system is designed to accommodate up to four 9830A Calculators. Only one calculator, however, can access the mass memory at any one instant. A 9866A Printer, or any other printer, is required for each system.

11273B INTERFACE KIT

The 11273B Interface Kit consists of a Mass Memory ROM, an interface cable, and a system tape cassette. The ROM contains some of the system commands needed to operate the mass memory drive. When the ROM block is plugged in, it uses 300 words of user read/write memory. The interface cable connects the calculator to the controller.

Finally, the mass memory system tape cassette contains programs to test the mass memory system, initialize new platters and duplicate the information contained on one platter to another platter. The system tape cassette also contains the system's 'bootstraps'. These are the remainder of the system commands and statements needed to operate the mass memory. They are loaded onto each mass memory platter during the initialization process. Once the bootstraps are loaded onto a platter, the bootstrap commands and statements are automatically transferred into the 300 word area mentioned above when needed (i.e., when a program or a user requires them).

One interface kit is required for each calculator connected to the controller.

11305A CONTROLLER

The controller interfaces the calculator(s) and the mass memory drive(s). It serves as a junction box for connecting one or more calculators and one or more mass memory drives. The 9868A I/O Expander cannot be used for this purpose; the 11305A Controller must be connected directly to the calculator(s). Of course, the 9868A I/O Expander can still be used to connect peripheral equipment to a calculator, whether that calculator is part of the mass memory system or disconnected from it.

12869A CARTRIDGE

A removable platter in its container is called a 'cartridge'. The platter is the heart of the system. As mentioned earlier, up to four platters at a time can be included in one system, each one of which has a storage capacity of 2.4 million bytes. Cartridges can be interchanged, of course, for an even greater storage capacity per system. Each platter must be initialized before using it for the first time. For this initialization procedure, refer to "Initializing New Platters", page 1-10.

9867A/B MASS MEMORY DRIVE

The 9867A Mass Memory Drive holds and operates one cartridge; the 9867B Mass Memory Drive holds and operates two platters. One of the two platters is permanently installed in the mass memory drive and the other (the cartridge) is removable. Several configurations are possible since you can incorporate up to four platters per system (e.g., two 9867B's, one 9867B and two 9867A's, four 9867A's).

NOTE

In order to operate the mass memory system, all drives connected to your controller must be switched ON. If you don't plan to access a drive, the LOAD switch may be set to its UNLOAD or OFF position, but power must still be on.

EQUIPMENT SUPPLIED

The following equipment is supplied with the 9880A or 9880B† Mass Memory and is necessary to operate the system:

Table 1-1. Equipment Supplied

Description	Quantity	-hp- Part Number
Controller	1	11305A
Mass Memory Drive †	1	9867A or 9867B
Interface Kit consists of:		
Mass Memory ROM	1	11273-67920
System Tape Cassette	1	11273-60002
Interface Cable Assembly	1	———
Operating Manual	1	09830-90008
Training Cassette	1	09830-90020
Quick Reference Card	2	09830-90021
Data Base Routines PAC	1	09830-76501
Cartridge	1	12869A
Cart	1 (Optional)	11304A
Fuse, 1.5 amp	3	2110-0043
Fuse, .75 amp	3	2110-0033
Power Cable	1	8120-1378 or 8120-1689

† The 9867A drive is supplied with the 9880A system, while the 9867B drive and a separate -hp- 13215A Power Supply are supplied with the 9880B system. Of course, additional drives can be purchased for use with either system.

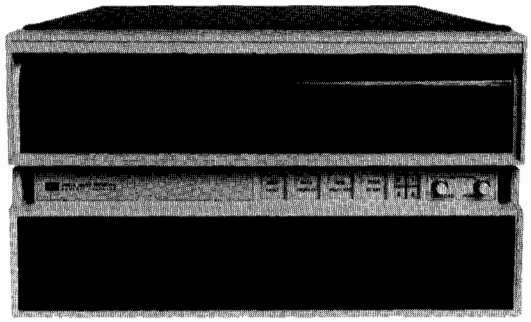


Figure 1–2. 9867A Mass Memory Drive

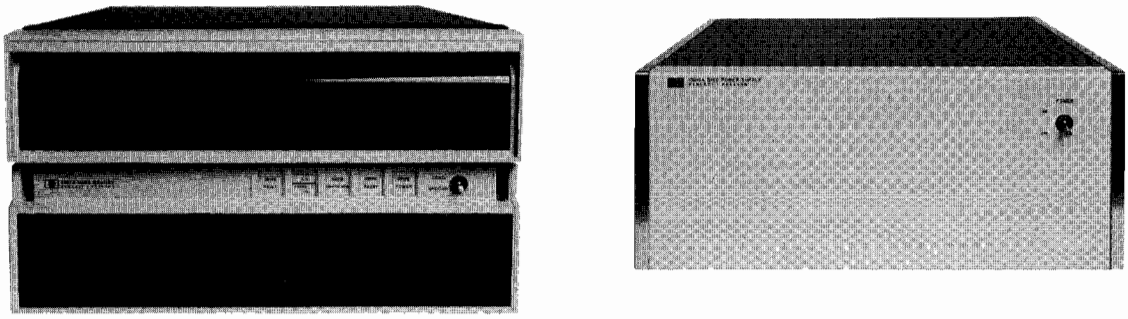


Figure 1-3. 9867B Mass Memory Drive with 13215A Power Supply

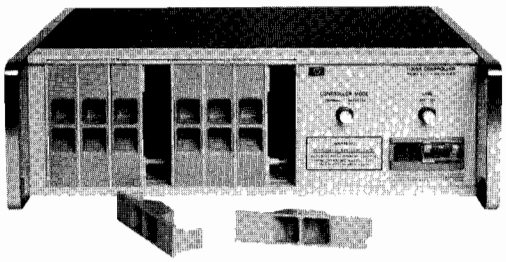


Figure 1-4. 11305A Controller

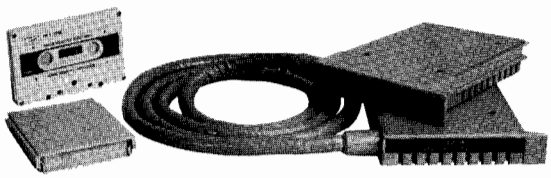


Figure 1-5. 11273B Interface Kit

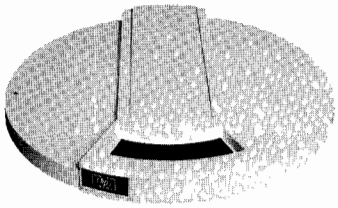


Figure 1-6. 12869A Cartridge

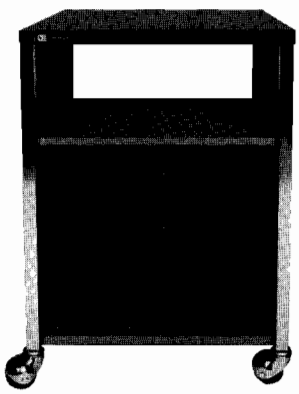


Figure 1-7. 11304A Cart

ADDITIONAL EQUIPMENT

You can use string and matrix commands for greater flexibility in the mass memory system. The matrix commands require the installation of an 11270 Matrix Operations ROM to the calculator and the string commands require an 11274 String Variables ROM.

Of course, all other Model 30 ROMs are compatible with your mass memory and can be plugged into the calculator in any but the top ROM slot when the Mass Memory ROM block is installed.

Once again, a 9866A Printer, or other printer, is required for operation of this system.

MULTIPLE DRIVE/CALCULATOR SYSTEMS

For purposes of instruction, it is assumed that you are working with one platter and one calculator. Specific information for operating more than one platter at a time is available in "MULTIPLE PLATTERS", page 4-4.

If more than one calculator is connected to the controller, an interface kit is required for each of them.

The calculator that is currently accessing a platter has control priority over any other calculator that attempts to access any platter in the system. The second calculator automatically accesses the platter it specifies when a delay of a half second or more occurs in the first calculator's access operation. Any platter can be accessed by any calculator connected to the controller, but two or more calculators cannot access the same or different platters simultaneously.

There are certain protection features built into the mass memory system and discussed in the following chapters. If these protection features are not used, one user can access, modify or even erase the information on another user's platter.

INSTALLATION AND TURN-ON PROCEDURES

An -hp- service representative will install and inspect the mass memory system for you initially; local sales and service offices are listed at the back of this manual.

CAUTION

SERIOUS DAMAGE CAN RESULT IF THE 9867A/B MASS MEMORY DRIVE IS MOVED ABRUPTLY. DESPITE ITS SIZE, THE MASS MEMORY DRIVE IS AN EXTREMELY DELICATE INSTRUMENT AND MUST BE HANDLED CAREFULLY AT ALL TIMES.

INSTALLING THE PLUG–IN ROM BLOCK

The complete procedure to install a plug-in ROM block is in the Operating and Programming Manual for the 9830A Calculator. Following are some reminders; please note the changes from the standard procedure:

- Switch the calculator OFF before installing or removing a ROM.
- The Mass Memory ROM *must* be installed in the *top* ROM slot behind the ROM door on the left side of the calculator.
- The label on the ROM should be right-side-up and facing the ROM door when the ROM is properly installed.
- Ensure that the ROM is properly mated to the connector at the back of the slot before switching the calculator ON.

NOTE

Use of the calculator with the plug-in ROM, but without the loaded bootstraps, may result in apparently illogical error messages. DISP and PRINT statements, for example, cannot be executed. For this reason, remove the Mass Memory ROM whenever the interface assembly is disconnected from the rear of the calculator or the mass memory drive's power switch is turned OFF.

TURN–ON

Following is a list of indicators which are part of the 9867A or 9867B Mass Memory Drive. When lit, they indicate the conditions shown below. Parenthetical numbers refer to the specific drive model.

DRIVE FAULT –	Malfunction in hardware or a read-write head movement operation was not completed within 850 milliseconds.
DATA PROTECT (9867A) –	Data protect switch is in the ON position. When lit, the cartridge is protected against any write operations.
L/D PROTECT (9867B) –	L/D protect switch is in the ON position. When lit, the lower, fixed platter is protected against any write operations.
U/D PROTECT (9867B) –	U/D protect switch is in the ON position. When lit, the upper, removable cartridge is protected against any write operations.
DOOR UNLOCKED –	LOAD switch is in the OFF or UNLOAD position and the drive spindle is stopped.
DRIVE READY –	Drive spindle motor has reached 2400 RPM, the air filtration system has been purged of unclean air, and the read-write heads are in a loaded position over the platter. Stays lit during mass memory operations.
UNIT SELECT INDICATOR – (9867A)	Indicates the unit number of the platter assigned to the mass memory drive.

INSTALLATION AND TURN-ON PROCEDURES

(Continued)

The procedure shown below must be followed to turn your mass memory system on. It is to be followed only after your -hp- service representative connects the mass memory components. This procedure is also performed, initially, by the service representative who installs your system.

- Once the Mass Memory ROM is installed in the calculator, switch the calculator and printing device ON.
- Set the mass memory drive LOAD switch to the OFF or UNLOAD position and press the POWER switch of the drive ON (its 'in' or 'up' position). All drives connected to your controller must be switched on in this manner before the system can be operated.
- Wait until the light marked DOOR UNLOCKED is turned on. This will take approximately 5 seconds.† Then turn the LOAD switch of the drive(s) to the ON or LOAD position.
- The DOOR UNLOCKED indicator will go off. Wait until the light marked DRIVE READY is turned on. This will take approximately 25 seconds. Then switch the controller on by pressing the LINE button of the controller ON. Be sure the intake fan at the back of the controller is not blocked. Air must circulate freely. *The controller must be the last component to be switched on.*††
- Set the LOAD switch to the UNLOAD or OFF position on each drive which you don't plan to access.

CAUTION

IF, AT ANY TIME DURING MASS MEMORY OPERATION, YOU HAVE REASON TO BELIEVE THE DRIVE IS NOT WORKING PROPERLY, TURN THE **LOAD** SWITCH OF THE DRIVE TO THE **OFF** OR **UNLOAD** POSITION FIRST. WHEN THE LIGHT MARKED **DOOR UNLOCKED** IS ON, TURN OFF ALL OF THE COMPONENTS AND CALL YOUR LOCAL -hp- SALES AND SERVICE OFFICE. DO NOT ATTEMPT TO RETRIEVE DATA STORED ON A CARTRIDGE BY USING IT IN ANOTHER DRIVE. DAMAGE MAY RESULT TO THE SECOND DRIVE.

† This procedure assumes a removable cartridge is in your mass memory drive. (A 9867B also must have a cartridge inserted at all times for system operation.) If your cartridge is not in the drive, wait until the DOOR UNLOCKED light is on, then open the front door of the drive by pulling out and down from the upper edge of the front door. Install the cartridge carefully, with the -hp- label right side up and facing you. Do not force the cartridge into the drive, as this may damage the read-write floating heads. Close the front door of the drive.

†† The controller should never be turned on while a 9830 program is running.

TURN–OFF

To turn off the mass memory system, reverse the steps in the above procedure.

CAUTION

WHEN TURNING THE MASS MEMORY SYSTEM OFF, TURN THE **LOAD** SWITCH TO THE **OFF** OR **UNLOAD** POSITION AND WAIT FOR THE **DOOR UNLOCKED** INDICATOR TO LIGHT BEFORE PRESSING THE DRIVE **POWER** SWITCH OFF. PROPER HEAD UNLOADING IS GUARANTEED ONLY IF THE **LOAD** SWITCH IS OPERATED WITH POWER **ON**. HEAD OR PLATTER DAMAGE MAY OCCUR UNLESS THIS PROCEDURE IS FOLLOWED IN THE CORRECT ORDER.

UNIT SELECT AND DATA PROTECT SWITCHES

In the 9867A Mass Memory Drive, the unit select switch and the data protect switch are located beneath the cartridge inside the front door. The unit select switch for the 9867B drive is internal and is set by the -hp- representative who initializes your system. The numbers 0 or 2 are assigned to the upper cartridge, while 1 or 3 designate the corresponding lower platter.

There are two data protect switches for the 9867B; one is for the upper, removable cartridge and the other is for the lower platter. Both of these switches are located beneath the cartridge inside the front door. In the ON position, the data protect switch prevents write operations (PRINT#, MAT PRINT#, SAVE, OPEN, KILL and PRO) on the the specified platter.

INSTALLATION AND TURN-ON PROCEDURES

(Continued)

INITIALIZING NEW PLATTERS

Initialization defines the tracks on the platter so that they may be referenced by the system. The system tape cassette contains bootstraps which will be loaded on each platter (one for a 9867A and two for a 9867B) of your mass memory system initially by an -hp- service representative. Each new platter (fixed or removable) must be initialized in this manner before it can be used with your system. The initialization takes about one hour per platter. Subsequent loading of the bootstraps on each new platter is accomplished according to the following procedure. Please note that, once the cassette is loaded onto a platter, it remains there. This procedure does *not* have to be repeated each time the mass memory system is turned on.

- Once the Mass Memory ROM is installed in the calculator, switch the calculator and printing device ON.
- Set the mass memory drive LOAD switch to the OFF or UNLOAD position and press the POWER switch of the drive ON (its 'in' or 'up' position).
- Wait until the light marked DOOR UNLOCKED is turned on. This will take approximately 5 seconds.
- Open the front door of the drive by pulling out and down from the upper edge of the front door.
- Install the cartridge carefully, with the HP label right side up and facing you. *Do not force the cartridge into the drive.*
- Close the front door of the drive.
- Press the LOAD switch of the drive to the ON or LOAD position.
- The DOOR UNLOCKED indicator will go off. Wait until the light marked DRIVE READY is turned on. This will take approximately 25 seconds. Then switch the controller on by presssing the LINE button of the controller ON.

CAUTION

IF, AT ANY TIME DURING MASS MEMORY OPERATION, YOU HAVE REASON TO BELIEVE THE DRIVE IS NOT WORKING PROPERLY, TURN THE **LOAD** SWITCH OF THE DRIVE TO THE **OFF** OR **UNLOAD** POSITION FIRST. WHEN THE LIGHT MARKED **DOOR UNLOCKED** IS ON, TURN OFF ALL OF THE COMPONENTS AND CALL YOUR LOCAL -hp- SALES AND SERVICE OFFICE. DO NOT ATTEMPT TO RETRIEVE DATA STORED ON A CARTRIDGE BY USING IT IN ANOTHER DRIVE. DAMAGE MAY RESULT TO THE SECOND DRIVE.

The initialization procedure is accomplished in two phases. Once all components of the mass memory system are switched on, follow the procedure below to load the mass memory system tape cassette bootstraps onto your platter.

1. Insert the mass memory system tape cassette in the tape transport of the calculator and close the transport door.
2. Key in and execute: LOAD BIN 60.
3. When the cassette file has been loaded into the calculator memory and control returns to the calculator, key in and execute the word, INITIALIZE.

From this point on, your calculator printer will instruct you. The first message printed is:

```
SET CONTROLLER MODE SWITCH TO INITIALIZE.

WHEN DONE, PRESS CONT, EXECUTE.
```

4. The button next to the LINE button on the controller, marked CONTROLLER MODE, should be switched to INITIALIZE (its 'in' position).
5. Key in and execute: CONT from the keyboard.

The next message printed is:

```
INPUT NUMBER OF THE UNIT TO BE INITIALIZED.
```

6. Input and execute the unit number of the platter you wish initialized. (Remember: On the 9867B drive, the upper, removable cartridge is number 0 or 2, while the lower, fixed platter is correspondingly 1 or 3.)

The next message printed is:

```
UNIT NUMBER = N. OK? INPUT 1 FOR YES, 0 FOR NO.
```

where 'N' is the unit number you specified.

7. If correct, key in and execute: 1. (A 0 executed from the keyboard will return you to the point at which you switched the INITIALIZE button ON. Continue this procedure from step #5.)

The next message printed is:

```
INITIALIZATION PHASE 1 WILL TAKE APPROXIMATELY 3 MINUTES.
```

NOTE

If an ERROR† occurs any time during execution of the initialization procedure, re-execute the instructions from step #1 and follow the procedure again. If the same error message is displayed again, call your local -hp- sales and service office for help.

After phase 1 is completed, the next message printed is:

```
SET CONTROLLER MODE SWITCH TO NORMAL.
```

8. Set the CONTROLLER MODE button to NORMAL (its 'out' position).
9. Key in and execute: CONT from the keyboard.

† ERROR 900 Defective track in system area (tracks 0-7, 404, 405) of platter, or 7 or more defective tracks in user area.
ERROR 901 Transfer of information between calculator and platter is not operating normally.
ERROR 902 Bootstraps are not verified correctly.

INSTALLATION AND TURN-ON PROCEDURES

(Continued)

The next message printed is:

```
INITIALIZATION PHASE 2 WILL TAKE APPROXIMATELY 1 HOUR.
```

At this point, a track-by-track verification routine is executed. This second phase is automatic, while the display flashes:

```
INIT
```

The next message printed is:

```
DEFECTIVE TRACKS:
```

When phase 2 is completed, a list of defective tracks on your platter is printed. If there are no defective tracks, NONE is printed. The next message printed is:

```
THE MASS MEMORY BOOTSTRAPS HAVE BEEN LOADED AND VERIFIED.
DO YOU HAVE ANOTHER PLATTER (OR UNIT) TO BE INITIALIZED?
IF YES, TYPE INITIALIZE, PRESS EXECUTE.
IF NO, PRESS CONT, EXECUTE.
```

If another platter is to be initialized, key in and execute the word, INITIALIZE. Follow the previous procedure from step #4. Pressing CONT EXECUTE initializes the calculator. Control is returned.

LOADING AND VERIFYING BOOTSTRAPS

Once a platter has been initialized, bootstraps can be loaded onto it or verified at any time, without erasing information already on the platter. An ERROR 902 message may be avoided by loading the bootstraps again before verifying that they were loaded correctly.

To load the bootstraps on an initialized platter, perform the following procedure:

- Key in and execute: LOAD BIN 2, with the mass memory system tape cassette in your calculator tape transport.
- When control returns, key in and execute: UNIT N (where N is the unit number of the platter you wish to load). This step can be ignored if your platter is designated as unit 0.
- Key in the word, BOOT, and execute it.

The tape cassette begins loading information onto the platter. After about three minutes, control returns to the calculator. At this point, the bootstraps are loaded onto your platter.

To verify that the bootstraps were loaded correctly, perform the following procedure:

- Key in and execute: LOAD BIN 3, with the mass memory system tape cassette in your calculator tape transport.
- When control returns, key in and execute: UNIT N (where N is the unit number of the platter you wish to verify). This step can be ignored if your platter is designated as a unit 0.
- Key in the word, VERIFY, and execute it.

The tape cassette begins verifying information on your platter. After about three minutes, control returns to the calculator and the tape is rewound automatically. Verification of your bootstraps has been accomplished.

SYSTEM TEST INSTRUCTIONS

To test the operation of your mass memory system, be sure the Mass Memory ROM is inserted in the top ROM slot of the calculator. Also, insert the ROMs you plan to use with the mass memory system (i.e., String Variables ROM, Matrix Operations ROM, or both – other ROMs can also be inserted). These ROMs can be inserted in any ROM slot except the top one, reserved for the Mass Memory ROM.

Make sure the mass memory system components are connected properly and switched ON. An initialized cartridge must be in the drive unit. If a 9867B drive is to be tested, its fixed platter must be initialized also.

Insert the mass memory system tape cassette in the tape transport of the calculator and close the transport door. Press SCRATCH A EXECUTE.

For an explanation of the system test, before it is executed, press LOAD EXECUTE. After control returns to the calculator, press RUN EXECUTE. As the test is run, files are created and take up available user space on your platter. The calculator prints the information on the next page and begins running the test automatically.

(Continued)

```
                    MASS MEMORY EXERCISER
                    ---- ------ --------

THIS EXERCISER USES THE FOLLOWING FILE NAMES AND SPACE.

                NAME        SPACE(RECORDS)
                ----        --------------
                HP++++          3
                HP+++/          3
                HP+++-          3
                HP+++*          2     (STRING ROM)
                HP++*/          2     (MATRIX ROM)
                HP++**          2
                HP++++          1
                HP+***          2
                HP/+**          1
                HP/+/+          8
                HP/+/-          8
                HP/+/*          8
                HP/+//          8
                HP////      HP/+/- RENAMED

IF AN ERROR 97 OCCURS, YOU HAVE USED ONE OF THE ABOVE RESERVED
NAMES.


THE EXERCISER USES THE FOLLOWING MASS MEMORY COMMANDS:
   UNIT                                  PRINT#
   OPEN                                  READ#
   KILL(NO PROTECTION CODE)              IF END#
   FILES                                 TYP(F),TYP(-F)
   ASSIGN(NO PROTECTION CODE)            DCOPY
   SAVE                                  DREN(NO PROTECTION CODE)
   GET                                   SAVE KEY
   CHAIN                                 GET KEY

OPTIONAL WITH STRING ROM              OPTIONAL WITH MATRIX ROM
   DBYTE                                 MAT PRINT#
   DEXP                                  MAT READ#
   PRINT#
   READ#


THE EXERCISER DOES NOT USE THE FOLLOWING COMMANDS:
   CATALOG                               PROTECT
   DAVTP                                 DGET
   DFDUMP                                DFLOAD
```

To run the test without an explanation, key in and execute: LOAD 70. When control returns to the calculator, press RUN EXECUTE. Regardless of whether an explanation is printed, the first question displayed on the calculator is:

```
WHAT IS THE UNIT# TO BE TESTED?
```

Key in the unit number (0-3) you will be working with and press EXECUTE.

The next question displayed is:

```
STRING ROM INSTALLED? 1=YES;0=NO
```

If a String Variables ROM is inserted in your calculator, key in 1 and press EXECUTE. If you don't have a String Variables ROM, key in 0 and press EXECUTE.

The next question displayed is:

```
MATRIX ROM INSTALLED? 1=YES;0=NO
```

If a Matrix Operations ROM is inserted in your calculator, key in 1 and press EXECUTE. If you don't have a Matrix Operations ROM, key in 0 and press EXECUTE.

The final question displayed is:

```
NO. OF TIMES TO RUN TEST =?
```

Key in the number of times you want the test run. (To test the system completely one time, press 1.) Press EXECUTE.

At this point, the test is run. With both the String Variables ROM and the Matrix Operations ROM installed, one test takes approximately 9 minutes; the test takes approximately 15 minutes to run twice. After each test, a printed message informs you how many times the test has run. Multiples of five tests, however, are indicated by the following message:

```
TESTING ACCOMPLISHED N TIME(S)
```

When all tests are completed, the following message is printed and displayed:

```
TESTING COMPLETED N TIME(S)
```

The tape cassette is rewound automatically at the conclusion of these tests.

INSTALLATION AND TURN-ON PROCEDURES

(Continued)

During the course of the tests, the following displays occur with the frequency shown below:

MESSAGE	FREQUENCY	
DATA FILES OPENED	1	
CALCULATING NUMBERS	36	
SERIAL PRINT	36	
RANDOM PRINT	3	
I'M READING SERIALLY!	6	Combination repeated 4 times
I'M CHECKING READ ACCURACY	6	
SERIAL READ ONE DONE	1	
I'M READING SERIALLY!	6	Combination repeated 4 times
I'M CHECKING READ ACCURACY	6	
I'M READING RANDOMLY	4	
RANDOM READS COMPLETE	1	
I'M PRINTING MATRICES!	1	
I'M READING MATRICES!	1	Combination repeated 4 times
I'M CHECKING MATRICES!	6	
MATRIX READS COMPLETE	1	
I'M PRINTING STRINGS!	18	
STRING PRINT COMPLETE!	1	
I'M READING STRINGS!	24	
STRING READS COMPLETE!	1	
END OF FILE FOUND ON FILE 4	1	
I'M CHECKING KEYS!	3	
I'M FINISHED CHECKING KEYS!	1	

The messages displayed differ, according to the ROMs installed in the calculator. Obviously, the displays which mention MATRIX or MATRICES are only seen if the Matrix Operations ROM is installed, while displays which mention STRING or STRINGS require the String Variables ROM. In addition, the message, END OF FILE FOUND ON FILE 4, is displayed only when a Matrix Operations ROM is installed and the String Variables ROM is not.

Once the test is begun, do not rewind or remove the tape cassette. The last section of the test, using Special Function keys, is only run once, no matter how many times you specify the rest of the test should be executed. Consequently, removing the tape cassette before it is automatically rewound by-passes this section of the test and results in an error message.

MAINTENANCE REQUIREMENTS

As mentioned before, the mass memory system – especially the mass memory drive – is extremely delicate. If it is unintentionally bumped, damage may result. Even the most elaborate protection plans are not foolproof. For this reason, data back-up systems are strongly advised.

Make duplicate platters or tape cassette recordings of all your important files. When working with data that is updated regularly (e.g., payroll accounting), use a 'leap-frog' technique of updating this data, if possible. That is, record every other data transaction on alternate platters so that, if a hardware malfunction *does* occur, you are never more than one time-period out of date.

The importance of a clean environment in which to operate the mass memory system cannot be overemphasized. The platter rotates at high speed and the read-write heads of the drive actually fly over the surfaces of the platter, never contacting it. The flying altitude is only two microns over the platter surface – only 1/30th the diameter of a human hair – so it is important to keep dust particles and other contaminants that can impede head movement out of the system.

The read-write heads must be cleaned and the air filter changed *at least* every 60 days to ensure proper operation of the system. For this reason, purchase of a service contract with the mass memory system is highly recommended. Besides cleaning the heads and replacing air filters, your -hp- service representative checks head alignment to ensure compatibility of cartridges recorded in other systems and in your own system.

Finally, due to wear and tear, small anomalies occur on the surface of each platter. These platters should be overhauled after every six months of normal use. Call your local -hp- sales and service office for the name of the organization in your area that can perform this service.

After a period of months, weeks, or even days, it will become apparent that the information on your platter(s) is more valuable to you than is the entire mass memory system. Needless to say, with proper care of the system and regular check-ups, you will never have to worry about losing your valuable data.

CAUTION

IF, AT ANY TIME DURING MASS MEMORY OPERATION, YOU HAVE REASON TO BELIEVE THE DRIVE IS NOT WORKING PROPERLY, TURN THE **LOAD** SWITCH OF THE DRIVE TO THE **OFF** OR **UNLOAD** POSITION FIRST. WHEN THE LIGHT MARKED **DOOR UNLOCKED** IS ON, TURN OFF ALL OF THE COMPONENTS AND CALL YOUR LOCAL -hp- SALES AND SERVICE OFFICE. DO NOT ATTEMPT TO RETRIEVE DATA STORED ON A CARTRIDGE BY USING IT IN ANOTHER DRIVE. DAMAGE MAY RESULT TO THE SECOND DRIVE.

NOTES

Chapter 2

PROGRAM FILE OPERATIONS

Mass Memory files can be used to hold data (data files), programs (program files), or Special Function keys (key files). This chapter discusses program and key files and the commands which are used to manipulate them. For purposes of instruction, it is assumed that you are working with one platter and one calculator. Specific information for operating more than one platter at a time is available in "MULTIPLE PLATTERS", page 4-4.

The conventions shown below appear in the BASIC syntaxes.

brackets [] — items enclosed within brackets are optional.

coloring — colored items must appear as shown.

All programming statements must be preceded by a line number. Optional line number parameters indicate that the statement can be executed from the keyboard.

Following is an example using the complex version of the SAVE command, which is discussed in detail on page 2-3. Once you understand this syntax, you should have no trouble with any of the syntaxes in this manual.

Syntax:

[line number] SAVE "file name" [,1st line number [,2nd line number]]

Consider the syntax, step by step, from left to right.

1. The 'line number' parameter is optional. If included, the command can be executed from a program; without a line number, this command can be executed only from the keyboard.
2. The word, SAVE, in brown, is necessary to identify the command.
3. The 'file name' parameter is needed to specify the file that will record the program. Note that the quotation marks, in brown, enclosing the file name are necessary.
4. The '1st line number' parameter is optional. If it is specified, however, a comma is needed to separate the file name from the 1st line number.
5. The '2nd line number' parameter is also optional; for it to be specified, the 1st line number must also be specified; if both are specified, a comma is required to separate the two items.

Since the brackets are nested in this command, for the most deeply nested item to be specified, all other items must also be specified. If, however, the brackets had appeared as:

```
                    1     2
----- ----- [----] [----]
```

then this dependency between items would not exist; that is, the information within the second bracket could be specified without having the other bracketed information specified.

The syntax requirements for each command, statement and function are shown individually throughout this manual and are shown all together in Appendix E of this manual.

The terminology used in the following pages is shown below to help you understand the syntax of mass memory system statements.

file name – the name used to define a specific file. This name can contain up to six characters, with the following restrictions:

- no quotation marks (") in the name
- no commas (,) in the name
- no blanks (i.e., spaces) in the name
- a single asterisk (*) should not be used because that single character has special significance (see page 3-6).

Unless otherwise specified, the file name may be a string variable.

1st line number – the number that references the position of a statement in the program designated. This number must be an integer; no variable or expression is allowed.

2nd line number – the number that references the position of a statement in the program designated. This number can appear only if the 1st line number parameter is designated. The 2nd line number must be an integer; no variable or expression is allowed.

protection code – the combination of the characters used to protect a file. This code can contain up to six characters, but no quotation marks (") are allowed in the code.

Unless otherwise specified, the protection code may be a string variable.

When a parameter can consist of a string variable, two syntaxes are shown for the statement. Note that the second syntax, which includes a string variable in place of a given parameter, omits the quotation marks surrounding that parameter. The string variable syntax is valid only when the String Variables ROM is installed in the calculator. String variables, which may be used in the protection code, can consist of up to six characters including blanks and commas between characters; for use in the file name, however, blanks, commas and quotation marks cannot be used. Null and blank strings are not allowed.

There are eight commands which are used most often when working with program files.

SAVE	PROTECT
GET	KILL
CATALOG	CHAIN
SAVE KEY	GET KEY

The PROTECT and CATALOG commands can be executed only from the keyboard. All of the other commands (above) can be executed either from the keyboard or from a program.

PROGRAM COMMANDS

The commands described on the following pages are used to store and retrieve programs, kill and protect program files, catalog those files, and store and retrieve Special Function keys.

SAVE COMMAND

The SAVE command stores an entire program or part of it onto a specified file. This command parallels the STORE command of the Model 30 tape cassette instructions.

Syntax:

[line number] SAVE "file name" [,1st line number [,2nd line number]]
[line number] SAVE file name [,1st line number [,2nd line number]]

Examples:

```
10 SAVE "MONEY"
20 SAVE "FACE",50
   SAVE G$,50,150
```

With a program in the calculator's memory, you need only execute the SAVE command and the program will be stored on the platter. The specified program file takes up only the space needed for the program.

The file name must be unique. If you attempt to save a program using a name that has been used before for a data, program, or key file already on the platter, the SAVE command is rejected and ERROR 97 is displayed.

NOTE

All file names that begin with the characters, HP, and are followed by any combination of the characters, +, –, /, * (e.g., HP++/–, HP*+– –, etc.) should not be used. These names are used and subsequently erased by certain system programs. Aside from the restrictions mentioned on page 2-2, these are the only file names not allowed when operating the mass memory system.

The optional line number parameters enable you to store part of your program rather than all of it. With one line number specified, the SAVE command stores only the lines after and including the specified line. With both parameters, the lines between and including the specified lines are stored onto the platter. Of course, the whole program is still present in the calculator, whether all or only a portion of it is saved on the platter.

Key in the following program.

```
10 INPUT J
20 FOR I=J TO J+10
30 PRINT I,I↑2
40 NEXT I
50 END
```

PROGRAM COMMANDS

(Continued)

You can store this program on a file named, COUNT, for example, by executing this command from the keyboard:

```
SAVE "COUNT"
```

In the following program, a squaring program is tacked on to the original program, now stored in the file, COUNT.

```
10 REM  THIS IS THE COUNTING PORTION OF THE PROGRAM.
20 REM  IN ORDER TO SQUARE AND COUNT NUMBERS, ENTER #100 FIRST;
30 REM   THEN ENTER THE NUMBER YOU WANT SQUARED.
40 DISP "ENTER AN INTEGER";
50 INPUT J
60 IF J>99 THEN 120
70 FOR I=J TO J+10
80 PRINT I
90 NEXT I
100 END
110 REM  THIS IS THE SQUARING PART OF THE PROGRAM, WHICH CAN STAND ALONE.
120 DISP "ENTER ANOTHER INTEGER";
130 INPUT K
140 FOR L=K TO K+10
150 PRINT L,L^2
160 NEXT L
170 END
```

Lines 10 through 100 print ten consecutive numbers from the first number entered. Lines 110 through 170 print ten consecutive numbers and their squares, from the second number entered.

Once this program has been keyed into the calculator, you can save it in its entirety on a file named, MASTER, for example, by executing this command from the keyboard:

```
SAVE "MASTER"
```

You can also store the second half of the program on a file named, SQUARE, for example, by executing this command from the keyboard:

```
SAVE "SQUARE",110
```

Finally, to store the first half of this program on a file named, FIRST, for example, execute the following command from the keyboard.

```
SAVE "FIRST",10,100
```

In this way, you now have three programs stored under different file names. Once a program is stored on a platter, it can be loaded back into the calculator by means of a GET command, discussed later.

After a program is stored in the mass memory system, it still remains in the calculator memory. When modifications must be made to the program, you can store the new, modified program in the mass memory. This is done by executing the SAVE command and using a different file name parameter. In order to store the new program in a file of the *same* name, the original file must first be erased. This is accomplished by means of the KILL command, discussed on page 2-12.

CATALOG COMMAND

The CATALOG command lists every file on the platter with which you are working. The listing also shows information about each file.

Syntax:

CAT

Obviously, a listing of the contents of a given platter will vary with the contents themselves. Consider the sample catalog listing below.

```
 (3) ← revision # of bootstraps  ORIGIN
                                              ABSOLUTE       CURRENT
NAME       TYPE      TRACK     RECORD     LENGTH(R)      LENGTH(W)

COUNT       P          8          0           1              22
MASTER      P          8          1           1              193
SQUARE      P          8          2           1              74
FIRST       P          8          3           1              119
```

Notice that the column labelled NAME in the catalog listing above includes the file names, COUNT, MASTER, SQUARE, and FIRST. The P's under the label TYPE signify that these files are program files. The letters D and K (for data and key files) may appear in this column. They are discussed later in this chapter (page 2-13) and in Chapter 3 (page 3-4).

The labels, TRACK and RECORD, designate the location of the particular file on your platter.† Since the order in which the files were created may differ, these numbers vary from system to system and from platter to platter.

Notice also for the program files listed, that the current length, in words, is specified. The file COUNT uses only 22 words of memory on the platter. ABSOLUTE LENGTH(R) indicates the number of physical records which were reserved for these programs. The mass memory system determines how much memory, in records, is required to store a program and rounds that number to the next higher whole record. In the case of the file, COUNT, since the number of words required is less than 256 (i.e., less than one physical record), the number of records marked, rounded to the next higher whole integer, is 1.

The CATALOG command is a keyboard operation only; it cannot be executed from a program. Use the CATALOG command at any time to list the contents of your platter.

† See Appendix A, page A-11, for a complete description of platter structure.

PROGRAM COMMANDS

(Continued)

GET COMMAND

The GET command loads a program from the platter to the calculator; the values of all variables not defined in a COM statement are undefined.† The GET command can also renumber the statements of the program and run it without further instructions. This command parallels the LOAD command of the Model 30 tape cassette instructions.

Syntax:

[line number] GET "file name" [,1st line number [,2nd line number]]

[line number] GET file name [,1st line number [,2nd line number]]

Examples:

```
10 GET "WITHIT"
   GET "OFF",50
   GET T$,1000,100
```

Whenever the GET command is executed, all program lines in the specified file are loaded into memory. All program lines previously in the calculator memory are erased unless the 1st line number is specified.

If the 1st line number is specified, the loaded program lines are renumbered with the beginning line number corresponding to the specified 1st line number. Program lines previously in memory, with line numbers lower than the 1st line number, are retained; all other lines previously in memory are erased.

In the calculator mode (after the program is loaded into memory):

- If the 2nd line number is not specified, the calculator halts.
- If the 2nd line number is specified, program execution begins at this line number.

† When programs are loaded into calculator memory from the platter by means of a GET command, any string variables currently in memory are lost, including those strings defined in a COM statement. (Numeric variables defined in a COM statement are, of course, retained.) Thus, if GET must be executed, then those strings which must be saved should first be stored on a separate file, so that they can be loaded back into memory afterwords. A CHAIN command (described below) retains string variables defined in a COM statement.

In the programming mode (in this case, specifically after the GET command is executed during program execution):

- If the 2nd line number is not specified, program execution is 'restarted' either with:
  - the program line immediately following the GET command in the original program, or with
  - the first line of the loaded program; that is, if there were no lines after the GET command in the original program, or if the lines were destroyed by the GET command.
- If the 2nd line number is specified, program execution is 'restarted' with this line number.

Suppose the program shown below was stored on your platter under the file named, MASTER.

```
10 REM  THIS IS THE COUNTING PORTION OF THE PROGRAM.
20 REM  IN ORDER TO SQUARE AND COUNT NUMBERS, ENTER #100 FIRST!
30 REM   THEN ENTER THE NUMBER YOU WANT SQUARED.
40 DISP "ENTER AN INTEGER";
50 INPUT J
60 IF J>99 THEN 120
70 FOR I=J TO J+10
80 PRINT I
90 NEXT I
100 END
110 REM  THIS IS THE SQUARING PART OF THE PROGRAM, WHICH CAN STAND ALONE.
120 DISP "ENTER ANOTHER INTEGER";
130 INPUT K
140 FOR L=K TO K+10
150 PRINT L,L↑2
160 NEXT L
170 END
```

The following command, executed from the keyboard, loads the program which is in file, MASTER into the calculator.

```
GET "MASTER"
```

The program can be altered once in the calculator, but the information on the platter remains unchanged.

The syntax below, in addition to loading the program into the calculator, renumbers the statements starting with line number 200.

```
GET "MASTER",200
```

PROGRAM COMMANDS

(Continued)

A listing of the calculator memory at this point is shown below.

```
10 REM  THIS IS THE COUNTING PORTION OF THE PROGRAM.
20 REM  IN ORDER TO SQUARE AND COUNT NUMBERS, ENTER #100 FIRST!
30 REM   THEN ENTER THE NUMBER YOU WANT SQUARED.
40 DISP "ENTER AN INTEGER"!
50 INPUT J
60 IF J>99 THEN 120
70 FOR I=J TO J+10
80 PRINT I
90 NEXT I
100 END
110 REM  THIS IS THE SQUARING PART OF THE PROGRAM, WHICH CAN STAND ALONE.
120 DISP "ENTER ANOTHER INTEGER"!
130 INPUT K
140 FOR L=K TO K+10
150 PRINT L,L↑2
160 NEXT L
170 END
200 REM  THIS IS THE COUNTING PORTION OF THE PROGRAM.
210 REM  IN ORDER TO SQUARE AND COUNT NUMBERS, ENTER #100 FIRST!
220 REM   THEN ENTER THE NUMBER YOU WANT SQUARED.
230 DISP "ENTER AN INTEGER"!
240 INPUT J
250 IF J>99 THEN 310
260 FOR I=J TO J+10
270 PRINT I
280 NEXT I
290 END
300 REM  THIS IS THE SQUARING PART OF THE PROGRAM, WHICH CAN STAND ALONE.
310 DISP "ENTER ANOTHER INTEGER"!
320 INPUT K
330 FOR L=K TO K+10
340 PRINT L,L↑2
350 PRINT I
360 END
```

As you can see, the program originally loaded has not been erased; the second program, beginning on line 200, has been loaded after it. Had the second program been renumbered from line 30, only lines 10 and 20 of the original program would have remained; all line numbers from 30 on would have been either printed over or erased.

The GET command can also be used to begin program execution immediately.

```
GET "MASTER",200,107
```

By executing the command above, the program is renumbered from line number 200 again, and executed beginning at line 107. Since line 107 does not exist in this particular program, execution begins at the first available line number greater than 107 — in this case, line 110. The GET command with both optional parameters is useful for applying certain initial criteria to various applications.

CHAIN COMMAND

The CHAIN command is identical to the GET command discussed previously, except that current values of variables are not erased. This command parallels the LINK command of the Model 30 tape cassette instructions.

Syntax:

[line number] CHAIN "file name" [,1st line number [,2nd line number]]

[line number] CHAIN file name [,1st line number [,2nd line number]]

Examples:

```
 10 CHAIN "GANG"
 20 CHAIN "REACT",230
    CHAIN Q$,500,100
```

The CHAIN command is most often used in a program to link a program previously stored on the platter.

The following three programs stored in files, BEGIN, MIDDLE and END, respectively, illustrate several ways in which the CHAIN command can be used.

"BEGIN" (First Program Segment)

```
10 PRINT "THE FIRST PROGRAM SEGMENT HAS BEEN EXECUTED."
20 PRINT
30 CHAIN "MIDDLE",10,10
40 END
```

"MIDDLE" (Second Program Segment)

```
10 J=0
20 GOTO 40
30 J=2
40 IF J=1 THEN 100
50 IF J=2 THEN 150
60 PRINT "THE SECOND PROGRAM SEGMENT HAS BEEN"
70 PRINT "    CHAINED TO THE FIRST AND EXECUTED FROM LINE 10."
80 PRINT
90 CHAIN "END",10,10
100 PRINT "THE SECOND PROGRAM SEGMENT HAS BEEN CHAINED"
110 PRINT "     FROM THE THIRD PROGRAM SEGMENT AND EXECUTED AGAIN,"
120 PRINT "      THIS TIME FROM LINE 40."
130 PRINT
140 CHAIN "END",190,30
150 PRINT "THE SECOND PROGRAM SEGMENT HAS BEEN CHAINED"
160 PRINT "     FROM THE THIRD PROGRAM SEGMENT FOR THE LAST TIME,"
170 PRINT "     AND EXECUTED FROM LINE 30."
180 PRINT
190 END
```

PROGRAM COMMANDS

(Continued)

"END" (Third Program Segment)

```
10 IF J=2 THEN 70
20 PRINT "THE THIRD PROGRAM SEGMENT HAS BEEN CHAINED FROM"
30 PRINT "     THE SECOND PROGRAM SEGMENT AND EXECUTED FROM LINE 10."
40 PRINT
50 J=1
60 CHAIN "MIDDLE",10,40
70 PRINT "THE THIRD PROGRAM SEGMENT HAS BEEN CHAINED FROM THE"
80 PRINT "     SECOND PROGRAM SEGMENT AND EXECUTED FROM LINE NUMBER 40."
90 PRINT
100 END
```

In the above program segments, the values of variable J are retained when chaining from program to program. In this way, selected portions of the last two program segments are run. These programs are run by executing the GET command.

```
GET "BEGIN",10,10
```

See the printout below.

```
THE FIRST PROGRAM SEGMENT HAS BEEN EXECUTED.

THE SECOND PROGRAM SEGMENT HAS BEEN
    CHAINED TO THE FIRST AND EXECUTED FROM LINE 10.

THE THIRD PROGRAM SEGMENT HAS BEEN CHAINED FROM
    THE SECOND PROGRAM SEGMENT AND EXECUTED FROM LINE 10.

THE SECOND PROGRAM SEGMENT HAS BEEN CHAINED
     FROM THE THIRD PROGRAM SEGMENT AND EXECUTED AGAIN,
      THIS TIME FROM LINE 40.

THE SECOND PROGRAM SEGMENT HAS BEEN CHAINED
     FROM THE THIRD PROGRAM SEGMENT FOR THE LAST TIME,
     AND EXECUTED FROM LINE 30.

THE THIRD PROGRAM SEGMENT HAS BEEN CHAINED FROM THE
    SECOND PROGRAM SEGMENT AND EXECUTED FROM LINE NUMBER 40.
```

Be careful when renumbering a program with a CHAIN or GET command in it. It is possible that, when renumbering a program either from the keyboard or from another program, either of the line number parameters will fall outside the range of the calculator's line numbers (i.e., less than 1 or greater than 9999). This causes ERROR 4.

PROTECT COMMAND

The PROTECT command prevents erasure of your program files without the proper protection code, by assigning a certain code to the specified file.

Syntax:

PRO "file name", "protection code"

Example:

```
PRO "YOU","ME"
```

A protected file is not 'secure' in the Model 30 sense of the word. The program stored on a protected file can be loaded into calculator memory, modified, listed and executed without specifying the file's protection code.

NOTE

You can still load a program file which is protected and list or run it. The protection code is only to prevent accidental erasure of the file; it does not 'secure' a file.

In order to secure a program in a file, key the program into the calculator, key in and execute SEC, and then store it on the platter with a SAVE command. The file that contains this secure program can be protected at this point with a PROTECT command. Now the program can be loaded into the calculator memory and executed, but it cannot be listed or modified. It can only be erased if you know its protection code.

Make sure the protection code you specify is not obvious, in order to avoid unauthorized erasure of your protected file. Keep a record of your protection codes so that you can't forget them. *A protected program file cannot be erased without its correct protection code.*†

A program file must first be created, before it can be protected. Use a separate PROTECT command, after executing the SAVE command, to protect a specific file with a protection code. The PROTECT command is a keyboard operation only; it cannot be executed from a program.

† If the protection code for a file is lost or forgotten, that file cannot be erased. Under these circumstances, you may want to have the -hp- factory 'unprotect' the file. (Contact an -hp- office for a cost estimate.)

PROGRAM COMMANDS

(Continued)

KILL COMMAND

The KILL command erases the named file from the platter and releases the space it occupied for further storage.

Syntax:

[line number] KILL "file name" [,"protection code"]

[line number] KILL file name [,protection code]

Examples:

```
10 KILL "UMPIRE"
20 KILL "ROY","123"
   KILL A$,R$
```

Use the KILL command to erase files and the information contained in them. A protected file cannot be killed unless the protection code is included in the syntax. Attempting to kill a protected file with an invalid protection code results in ERROR 92. (Incidentally, this error message occurs if you *include* a protection code in the KILL command where none is necessary.) This makes it almost impossible for an unauthorized person to erase your program.

On occasion, you may wish to modify a program which had been stored previously on the mass memory system. To use the original file name for the updated program, follow this procedure:

1. Load the original program from the platter into the calculator with a GET command.
2. Modify the program in the calculator, as required, using the calculator edit keys and the END OF LINE key. (At this point a listing of the program is strongly recommended.)
3. Execute a KILL command from the keyboard to erase the outdated file from the mass memory system. The new program is still in the calculator memory; KILL commands do not affect it.
4. Now execute a SAVE command, specifying the original file name, to store the modified program.

The modified version of your original program is now stored on the mass memory system, as well as in the calculator memory.

If more than one calculator is connected to your mass memory system, there is a possibility that someone will be using the file you are trying to kill. In this case, your KILL command is temporarily held until the other user is finished. (A delay of one half second or more is sufficient.) When your calculator regains control, the file is killed.

SAVE KEY COMMAND

The SAVE KEY command stores Special Function key definitions on a specified file. This command parallels the STORE KEY command of the Model 30 tape cassette instructions.

Syntax:

[line number] SAVE KEY "file name"

[line number] SAVE KEY file name

Examples:

```
10 SAVE  KEY "LOGRAM"
   SAVE  KEY J$
```

The information on all 20 Special Function keys is stored on the file at the same time. Since only key definitions and programs stored in keys are saved with this command, you need two files to save a program which uses Special Function keys: one to save the mainline program and one to save the Special Function key definitions.

A catalog listing of a platter in which Special Function key definitions are stored is indicated by the letter, K, in the column labelled TYPE.

GET KEY COMMAND

The GET KEY command loads Special Function key definitions from a specified file of the platter to the calculator Special Function keys. This command parallels the LOAD KEY command of the Model 30 tape cassette instructions.

Syntax:

[line number] GET KEY "file name"

[line number] GET KEY file name

Examples:

```
10 GET  KEY "LOWATT"
   GET  KEY S$
```

When GET KEY has been executed, the original information is returned to the Special Function keys. All of the Special Function keys can then perform the same operations they previously did before you saved them on the file.

In addition, when GET KEY is executed from the keyboard, previously defined variable values are saved, similar to the CHAIN command. † When executed from a program, however, GET KEY initializes all variables, similar to the GET command.

† When programs are reproduced into calculator memory from the platter by a means of a CHAIN command, all variables currently in memory are saved. If GET KEY is used to reproduce the program, however, any string variables currently in memory are lost, including those strings defined in a COM statement. (Numeric variables defined in a COM statement are, of course, retained.) Thus, if GET KEY must be executed, then those strings which must be saved should first be stored on a separate file, so that they can be loaded back into memory afterwards.

NOTES

Chapter 3

DATA FILE OPERATIONS

While Chapter 2 of this manual discusses mass memory program and key files, *this* chapter describes the commands, statements and functions which are useful when working with *data* files. Again, for purposes of instruction, it is assumed that you are working with one platter and one calculator. Specific information for operating more than one platter at a time is available in "MULTIPLE PLATTERS", page 4-4.

A thorough understanding of data file structure is necessary before you can use the mass memory system efficiently. For this reason, refer to Appendices A and B of this manual to clarify the concepts presented in this chapter. Appendix A, especially, should be read in conjunction with this chapter on data file operations.

The conventions shown below appear in the BASIC syntaxes.

brackets [] – items enclosed within brackets are optional.
coloring – colored items must appear as shown.

All programming statements must be preceded by a line number. Optional line number parameters indicate that the statement can be executed from the keyboard.

The syntax requirements for each command, statement and function are shown individually throughout this manual and are shown all together in Appendix E of this manual.

The terminology used in the following pages is shown below to help you understand the syntax of mass memory system statements.

file name – the name used to define a specific file. This name can contain up to six characters, with the following restrictions:

- no quotation marks (") in the name
- no commas (,) in the name
- no blanks (i.e., spaces) in the name
- a single asterisk (*) should not be used because that character has special significance (see page 3-6).

Unless otherwise specified, the file name may be a string variable.

number of records – the total number of records in a file. This parameter can be an expression, as well as an integer. Single– and multiple–line functions are not allowed.

protection code – the combination of characters used to protect a file. This code can contain up to six characters, but no quotation marks (") are allowed in the code.

Unless otherwise specified, the protection code may be a string variable.

file number – the number delegated to a file by a FILES statement. This number can be any integer (constant, variable or expression) from 1 through 10. A non-integer's rounded value is automatically used. Single– and multiple–line functions are not allowed.

return variable – the variable in an ASSIGN statement used to determine a file's status. This parameter can be a simple variable or an array variable, as defined in the Model 30 Operating and Programming Manual.

list – the characters designated in a PRINT# or READ# statement. This parameter can consist of alphanumeric variables or string variables. Numeric constants and expressions are also allowed in PRINT# statements. Single– and multiple–line functions are not allowed.

record number – the number which represents the location of a record in a specific file. This number can be any integer (constant, variable or expression) which does not exceed the number of records in the associated file. Single– and multiple–line functions are not allowed.

line number – the number that references the position of a statement in the program designated. This number must be an integer; no variable or expression is allowed.

When a parameter can consist of a string variable, two syntaxes are shown for the statement. Note that the second syntax, which includes a string variable in place of a given parameter, omits the quotation marks surrounding that parameter. The string variable syntax is valid only when the String Variables ROM is installed in the calculator. String variables, which may be used in the protection code, can consist of up to six characters including blanks and commas between characters; for use in the file name, however, blanks, commas and quotation marks cannot be used.

There are six fundamental statements and commands used most often when working with data files.

OPEN	FILES	KILL
CATALOG	ASSIGN	PROTECT

The following statements and function are used when reading and printing data.

PRINT#	READ#
IF END#	TYP

These are discussed as they apply to serial file access and then again as they apply to random file access. Some of these commands (CATALOG, PROTECT, KILL) are discussed in "PROGRAM COMMANDS", page 2-3, since they apply to both data and program files. If you have already read Chapter 2, you may want to review these commands briefly, as they work almost identically for program files and data files.

PROTECT and CATALOG can be executed only from the keyboard, while IF END# can be executed only from a program. All of the other commands, statements, and functions (above) can be executed either from the keyboard or from a program. Please pay particular attention to the syntaxes described in the following pages; the (#) character must often be included in mass memory statements and commands to differentiate them from the BASIC language described in the Model 30 Operating and Programming Manual.

FUNDAMENTAL DATA COMMANDS

The statements and commands described on the following pages are used to create, destroy and access data files.

OPEN COMMAND

The OPEN command creates a data file with a specified number of physical records, assigns it a name, and places a logical end of file (LEOF) marker in the first word of each record in the file.†

Syntax:

[line number] OPEN "file name", number of records

[line number] OPEN file name, number of records

Examples:

```
10 OPEN "SESAME",24
20 OPEN "UP",10*X
   OPEN J$,20
```

Before data can be printed onto a data file, that file must first be opened and its size specified.

Each data file must be assigned a unique name. If you attempt to open a data file using a name which has been used before for a data, program or key file already on the platter, the OPEN command is rejected and ERROR 97 is displayed.

NOTE

All file names that begin with the characters, HP, and are followed by any combination of the characters, +, –, /, * (e.g., HP++/–, HP*+– –, etc.) should not be used. These names are used and subsequently erased by certain system programs. Aside from the restrictions mentioned on page 3-1, these are the only file names not allowed when operating the mass memory system.

The size of a file may vary from a minimum of one physical record (256 words) to a maximum of 4752 records.††

The first statement in the sample programs used to store data (included in this chapter), is generally an OPEN command. The OPEN command is included only to remind you that data files must be opened before data can be printed on them. Often, it is most convenient to execute the OPEN command from the keyboard, rather than from a program, since an error message results when you run the same program twice. (The program attempts to create a previously opened data file.)

Once a file has been opened (i.e., created) and space has been reserved for it, you can use that file thereafter. It can be erased only with a KILL command, which is discussed later in this chapter (page 3-5).

† See Appendix A for a discussion of file structure, end of record (EOR) and end of file (EOF) markers.

†† See Appendix B for a detailed discussion on how to estimate file size.

FUNDAMENTAL DATA COMMANDS

(Continued)

CATALOG COMMAND

The CATALOG command lists every file on the platter with which you are working. The listing also shows information about each file.

Syntax:

CAT

Obviously, a listing of the contents of a given platter will vary. Consider the sample catalog listing below.

NAME	TYPE	ORIGIN TRACK	ORIGIN RECORD	ABSOLUTE LENGTH(R)	CURRENT LENGTH(W)
SESAME	D	354	11	24	
UP	D	356	11	40	
$$$$$	D	360	3	20	

In this sample listing, the column labelled NAME includes the name of each of the files currently on a platter. The D's under the label TYPE, signify that these files are data files. The letters P and K, which may appear in this column, are discussed in Chapter 2 (pages 2-5 and 2-13). TRACK and RECORD designate the location of the particular file on your platter.† Since the order in which the files were opened may differ, these numbers vary from system to system and from platter to platter. The column labelled ABSOLUTE LENGTH(R), shows the number of physical records you specified for the file. CURRENT LENGTH(W) is not applicable to data files; it is discussed in "CATALOG Command", page 2-5).

The CATALOG command is a keyboard operation only; it cannot be executed from a program. Use the CATALOG command at any time to list the contents of your platter.

PROTECT COMMAND

The PROTECT command prevents erasure of, or access to, your data files without the proper protection code, by assigning a certain code to the specified file.

Syntax:

PRO "file name", "protection code"

† See Appendix A, page A-11, for a complete description of platter structure.

Example:

```
PRO "YOU","ME"
```

Make sure the protection code you specify is not obvious, in order to avoid unauthorized access to your protected files. Keep a record of your protection codes so that you can't forget them. *A protected data file cannot be accessed or erased without its correct protection code.*†

A data file must first be opened, before it can be protected. Use a separate PROTECT command after executing the OPEN command, to protect a specific file with a protection code. The PROTECT command is a keyboard operation only; it cannot be executed from a program.

KILL COMMAND

The KILL command erases the named file from the platter and releases the space it occupied for further storage.

Syntax:

[line number] KILL "file name" [,"protection code"]

[line number] KILL file name [,protection code]

Examples:

```
10 KILL "UMPIRE"
20 KILL "ROY","123"
   KILL A$,R$
```

Use the KILL command to erase files and the data contained in them. A protected file cannot be killed unless the protection code is included in the syntax. Attempting to kill a protected file with an invalid protection code results in ERROR 92. (Incidentally, this error message occurs if you *include* a protection code in the KILL command where none is necessary.) This makes it almost impossible for an unauthorized person to access or erase your data.

If more than one calculator is connected to your mass memory system, there is a possibility that someone will be using the file you are trying to kill. In this case, your KILL command is temporarily held until the other user is finished. (A delay of one half second is sufficient.) When your calculator regains control, the file is killed.

† If the protection code for a file is lost or forgotten, that file cannot be erased. Under these circumstances, you may want to have the -hp- factory 'unprotect' the file. (Contact an -hp- office for a cost estimate.)

FUNDAMENTAL DATA COMMANDS

(Continued)

FILES STATEMENT

The FILES statement declares which data files are to be used. If a file specified in the FILES statement has not been previously opened, an error message is displayed upon execution of the FILES statement.

Syntax:

[line number] FILES file name or * [,file name or *] [,...]

Examples:

```
10 FILES A,B,*,*,*
   FILES TOM,MARCY,LAUCK,**,JIMC,CALIF
```

Up to ten file names can be listed in the FILES statement syntax. These files, however, cannot be protected files. Notice also that you cannot use quotation marks around each file name. String variables cannot be used as file names.

Single asterisks (*) can also be used in place of file names. They allow you to:

- reference protected files
- use strings for file names
- reserve space for future files

When using asterisks, file assignment is completed by means of the ASSIGN statement. (See next section, page 3-7.)

A new FILES statement obsoletes the previous FILES statement, so you can include as many of these statements as you need in a given program. Executing a GET or CHAIN command does not destroy the last FILES statement. A new FILES statement also resets all pointers.†

The files listed in the FILES statement syntax are assigned numbers in the order in which they appear. For example, in the following statement, GEORGE is file #1, DATA is file #2, etc.

```
10 FILES GEORGE,DATA,D1,D,*,JOE
```

The file numbers are convenient labels by which you can reference specific files in a program or from the keyboard. Their use is apparent in the discussion of subsequent statements. File assignments can be erased at the end of a program for greater data security by executing FILES *.

† A pointer keeps track of the data item currently being accessed. Its use is discussed in detail later in this chapter "SERIAL FILE ACCESS" (page 3-9) and in Appendix A.

ASSIGN STATEMENT

ASSIGN statements work in conjunction with a previous FILES statement. ASSIGN allows you to:

- assign a file name to a certain file position
- determine the status of a given file
- use a string variable for a file name
- declare a protected file

Syntax:

[line number] ASSIGN "file name", file number, return variable [,"protection code"]

[line number] ASSIGN file name, file number, return variable [,protection code]

Examples:

```
10 ASSIGN "ANAME",2,I,"UNIQUE"
20 ASSIGN "MENT",F,X[1,2]
30 ASSIGN R$,6,J,C$
   ASSIGN L$,M+2,W,"XYZ"
```

In the syntax above, the file number determines the position the file name is to take with reference to the previous FILES statement. Since a FILES statement can contain up to 10 file names or asterisks, the file number must be a positive integer from 1 through 10. Consider the following program segment.

```
10 FILES ABEGG,*,MEL,A,HAFORD,*
20 ASSIGN "CARD",2,K,"WELL"
```

In this example, the asterisk in the FILES statement (in the file #2 position) is assigned the file name, CARD. Of course, the data file, CARD must have been opened before the FILES statement can be executed. Additional ASSIGN statements can be placed later in the same program for the purpose of reassigning a different file name to any file position. An ASSIGN statement sets the pointer of the specified file to the first item of the first physical record.

You can use a return variable to determine the status of the file. Any variable can be used.

In the previous example, K is the return variable. Its value is determined during the ASSIGN execution and can be used any time in the program. The value of the return variable indicates certain conditions, as listed in the table below. The return variable must be checked to avoid later unrecoverable errors.

Return Variable	Meaning
0	file is available
3	file does not exist
4	file number is out of range (i.e., it refers to a FILES statement position that does not exist)

FUNDAMENTAL DATA COMMANDS

(Continued)

Example A

To see how this can be useful, refer to the following 'OPEN FILES' program.

```
10 REM  THIS PROGRAM IS USED TO OPEN NEW, UNPROTECTED DATA FILES.
20 DIM A$[6]
30 FILES *
40 DISP "ENTER FILE NAME";
50 INPUT A$
60 ASSIGN A$,1,X
70 IF X=3 THEN 100
80 DISP A$;" WAS OPENED.  NEW NAME";
90 GOTO 50
100 DISP "NUMBER OF RECORDS";
110 INPUT J
120 OPEN A$,J
130 PRINT A$;" IS NOW OPENED.  NUMBER OF RECORDS =";J
140 PRINT
150 GOTO 40
160 END
```

In this example, line 70 instructs the calculator to branch to line 100 if the file name you enter does not exist, and open the file. Without this IF statement, ERROR 97 occurs when you attempt to create a file which has been previously opened. Use of the return variable (in this case, X) avoids that error message.

Although a string variable name cannot be included in a FILES statement, *per se*, you can use a string variable file name by referencing any file position with a string variable in the ASSIGN statement. This is done in the previous program by the FILES and ASSIGN statements. Notice also that the ASSIGN statement (line 60) does not need to directly follow the FILES statement (line 30).

Finally, the ASSIGN statement can be used to allow a protected data file to be accessed. For example, consider the following program segment.

```
10 FILES JANET,*,JUDY,MARK
20 ASSIGN "MIMI",2,V,"SISTER"
 |
 |
 |
 |
 |
 |
300 ASSIGN "SUB",4,P
```

In this example, obviously the file MIMI had been previously protected with the protection code "SISTER". MIMI is assigned the second position in the FILES statement. The protection code of a protected file must be included in order to access it. Omitting the protection code results in ERROR 92 in this case.

Line 300 re–assigns the fourth position of the FILES statement (MARK) to the unprotected file named SUB. All references to file number 4 from this point (line 300) on in the program refer to file SUB, not file MARK. File number 4 will refer to SUB unless a subsequent ASSIGN statement changes it (or another FILES statement is executed), as line 300 changed the original FILES assignment.

SERIAL FILE ACCESS

For each data file declared, a file pointer keeps track of the data item currently being accessed. The pointer moves through the file as you store or retrieve data items. Data is printed or read consecutively from the position of the pointer, which is set at the beginning of the file when a FILES or ASSIGN statement for that file is executed.

Different syntaxes of the following serial file access statements are required for random file access (data printed or read in *specific* records of a file). They are discussed in "RANDOM FILE ACCESS", page 3-21.

SERIAL PRINT# STATEMENT

The serial PRINT# statement prints data in the form of variables, numbers, or strings of characters. Data is printed serially on the specified file after the last item previously read or printed, or at the beginning of the file.

Syntax:

[line number] PRINT# file number; list [,END]

[line number] PRINT# file number; END

Examples:

```
10 PRINT #1;A,B$
20 PRINT #I;A$,B,6.1,9,"DATA",A,END
   PRINT #8*J/2;X
   PRINT #3;END
```

In general, the length of the data list is limited by the length of the BASIC statement (80 characters), or by the size of the file.†

When a PRINT# statement is executed, a logical end of record (LEOR) marker is placed after the last data item specified. When a data list is included in the PRINT# statement, the optional parameter, END, is used to print a logical end of file (LEOF) marker at the end of the data list. This LEOF marker replaces the LEOR marker which is placed at the end of each data list automatically when a serial PRINT# statement is executed.

† See Appendix B, "STORAGE REQUIREMENTS".

SERIAL FILE ACCESS

(Continued)

Following is an example using the PRINT# statement to record five students' identification numbers and test grades.

```
10 OPEN "I.D.",1
20 OPEN "GRADES",1
30 FILES I.D.,GRADES
40 FOR I=1 TO 5
50 DISP "STUDENT'S I.D.#";
60 INPUT X
70 PRINT #1;X
80 DISP "WHAT IS THE NEXT TEST SCORE";
90 INPUT R
100 PRINT #2;R
110 NEXT I
120 END
```

Notice that no ASSIGN statement is used, since neither file is protected. Use this program to print the following identification numbers on the file, I.D. and the corresponding grades on the file, GRADES.

I.D. #	Grade
25009	91
54362	88
11243	99.5
64597	62
74532	89

In the above program, two separate files are used: one for the students' identification numbers and one for their grades. The information can be combined into one file in the following manner.

```
10 OPEN "SCORES",1
20 FILES SCORES
30 FOR I=1 TO 5
40 DISP "STUDENT'S I.D.#/GRADE";
50 INPUT X,R
60 PRINT #1;X,R
70 NEXT I
80 PRINT #1;END
90 END
```

Line 60 prints the names and test scores of the students on the file, SCORES. The data items (I.D. numbers and grades) are printed alternately. Line 80 places an LEOF marker when the five sets of data elements are input.

This LEOF marker can be printed over, but data on the other side of it cannot be read. In attempting to execute a serial READ# statement (see next section, page 3-12), an end of file condition is established as a result of encountering this LEOF marker. If a PRINT# statement that includes the optional END parameter, but no list of data elements, is executed at the *beginning* of a file, an LEOF marker is placed there. Data remaining in the file cannot be read in a serial manner.

In terms of serial READ# statements, therefore, the data beyond the point at which the LEOF marker is placed, is effectively erased. Some of this data may be accessed, however, by moving the pointer to the beginning of a subsequent record beyond the LEOF marker (see "Repositioning the Pointer", page 3–13) or in a random manner (see "RANDOM FILE ACCESS", page 3–21).

A String Variables ROM enables you to enter students' names, rather than I.D. numbers. The I.D. variable, X, is replaced by a string variable and the program prints string names as data on the file. Using the data shown below,

Name	Grade
Al Jackson	91
Jayne Lamfers	88
Mark Levy	99.5
Brian Smith	62
Pat Stohrer	89

key in and run the following program after installing a String Variables ROM in your calculator.

```
10 OPEN "CLASS",1
20 DIM N$[15]
30 FILES CLASS
40 FOR K=1 TO 5
50 DISP "STUDENT'S NAME";
60 INPUT N$
70 DISP "WHAT IS NEXT TEST SCORE";
80 INPUT Z
90 PRINT #1;N$,Z
100 NEXT K
110 PRINT #1;END
120 END
```

SERIAL FILE ACCESS

(Continued)

SERIAL READ# STATEMENT

The serial READ# statement reads numbers and strings into variables serially from the specified file, starting after the last item printed or read. Substrings are not allowed.

Syntax:

[line number] READ# file number; list

Examples:

```
10 READ #1;A,B
20 READ #6;A$,B$,A,A1,A2
   READ #F;X,Y,Z[3,1]
```

Before you can work with data which has been stored in a file, you must first read the data into the calculator. Remember that you are not erasing the data stored in the platter by reading it; data is merely copied into the variables specified. (This data can be updated and re-stored into the original file — without using a KILL command — or into a new file.)

Recall the program on page 3-10, used to print data on the files, I.D. and GRADES. To read the data from these files back into the calculator and print the information on the calculator printer, use the following program.

```
10 FILES I.D.,GRADES
20 PRINT "STUDENT I.D. NUMBER          GRADE"
30 PRINT
40 READ #1;S
50 READ #2;T
60 PRINT S," ",T
70 GOTO 40
80 END
```

In this program, the FILES statement serves two purposes; it references the file number parameters in the READ# statements (lines 40 and 50) and it resets the pointers to the beginning of both files before the READ# statements are executed.

Upon execution of this program, ERROR 99 IN LINE 40 is displayed because you attempt to read data after an LEOR marker is detected. This automatically causes the READ# statement to attempt to get data from the next physical record. At this point, it encounters an LEOF marker in the first word, which was placed there when the file was opened. It is actually detection of the LEOF marker which triggers ERROR 99 in this case.

Data printed on the file, CLASS (see the program on page 3-11) can also be read back into the calculator. Use the following program to print this data on the calculator printer.

```
10 DIM P$[15]
20 FILES CLASS
30 PRINT "NAME OF STUDENT                    GRADE"
40 PRINT
50 FOR D=1 TO 5
60 READ #1;P$,Y
70 PRINT P$," ",Y
80 NEXT D
90 END
```

Notice that the READ# statement must specify the types of data (data elements or string variables) in the order in which they were originally stored in the file. Line 60 reads a string variable and then a data point. This program can run only when the order of the data on file, CLASS is known.

The variables into which you read data items do not necessarily have to be the same variables from which you printed the data items on the file. Although the variable name changes (from N$ and Z, when stored, to P$ and Y, when retrieved), the order in which the two data types are accessed is the same.

If the FOR...NEXT loop is set for D = 1 to *6*, the READ# statement encounters the LEOF marker previously placed by the PRINT# 1; END statement. This marker replaced the LEOR marker at the end of the five sets of data items. Encountering the LEOF marker establishes an end of file condition which is discussed later in this section ("IF END# Statement", page 3-16).

REPOSITIONING THE POINTER

As mentioned earlier, a pointer is maintained by the mass memory system. The printer specifies where data storage or data retrieval begins. The pointer is automatically positioned at the beginning of the first physical record in a file after execution of a FILES statement or an ASSIGN statement. It is positioned at the next available storage location in the physical record after execution of a PRINT# statement. Finally, it is positioned at the next stored data item location of a physical record after execution of a READ# statement. The pointer is left unchanged in each file before execution of a serial PRINT# or READ# statement.

It is often necessary to position the pointer to the beginning of a specific physical record in a file before executing a serial READ# statement. The following variation of the READ# statement is used for this purpose.

Syntax:

[line number] READ# file number, record number

Examples:

```
10 READ #3,R
   READ #1,4*Y+7
```

SERIAL FILE ACCESS

(Continued)

When a record number is specified and the list of variables is not included in the serial READ# statement, no data is read. Instead, the pointer is repositioned to the beginning of the record specified. A serial PRINT# or READ# statement can be executed after the pointer has been repositioned, to access the beginning of the specified physical record, rather than the beginning of only the first record of the file.

To see how this works, first use the following program to store consecutive numbers beginning from the eleventh record of a 15-record file named DATA15.

```
10 OPEN "DATA15",15
20 FILES DATA15
30 I=1
40 READ #1,11
50 PRINT #1;I
60 I=I+1
70 GOTO 50
80 END
```

The FILES statement (line 20) sets the pointer to the beginning of the first record in the file. The pointer is simply repositioned to the beginning of the eleventh record of DATA15 by executing line 40.

After printing onto physical records 11-15 of DATA15, which takes about 25 seconds, ERROR 99 is displayed. This indicates that a physical end of file (PEOF) marker is encountered and no additional data can be printed in the file.

The following program is now used to read the data from the beginning of record 14.

```
10 FILES DATA15
20 READ #1,14
30 READ #1;A,B,C,D,E,F,G,H
40 PRINT A;B;C;D;E;F;G;H
50 GOTO 30
60 END
```

The FILES statement (line 10) automatically sets the pointer to the beginning of the first record. The pointer is simply repositioned to the beginning of the fourteenth record in DATA15 by executing line 20. The serial READ# statement begins reading data from that point on.

Since each full precision number uses four words of memory, 64 numbers can be printed onto a 256–word physical record.† On the file DATA15, the following numbers are stored on these corresponding records.

Record #	Numbers
1-10	none
11	1-64
12	65-128
13	129-192
14	193-256
15	257-320

The previous program reads the data on records 14 and 15 (i.e., numbers 193-320) and lists this information, eight numbers per row. The printout of this program is shown below.

```
 193   194   195   196   197   198   199   200
 201   202   203   204   205   206   207   208
 209   210   211   212   213   214   215   216
 217   218   219   220   221   222   223   224
 225   226   227   228   229   230   231   232
 233   234   235   236   237   238   239   240
 241   242   243   244   245   246   247   248
 249   250   251   252   253   254   255   256
 257   258   259   260   261   262   263   264
 265   266   267   268   269   270   271   272
 273   274   275   276   277   278   279   280
 281   282   283   284   285   286   287   288
 289   290   291   292   293   294   295   296
 297   298   299   300   301   302   303   304
 305   306   307   308   309   310   311   312
 313   314   315   316   317   318   319   320
```

ERROR 99 is displayed at this point, indicating a PEOF marker is detected and there is no more data to be read. This error message can be avoided by using the IF END# statement, discussed in the next section.

† See Appendix B for a detailed discussion on how to estimate file size.

SERIAL FILE ACCESS

(Continued)

IF END# STATEMENT

The IF END# statement sets up a condition in the program. If a PEOF marker is encountered during execution of a serial PRINT# or READ# statement, or if an LEOF marker is encountered during execution of a serial READ# statement, the program branches to the line number specified in the previous IF END# statement, thus avoiding an ERROR. This makes it possible to use a file whose exact contents are unknown.

Syntax:

line number IF END# file number THEN line number

Examples:

```
10 IF END#3 THEN 222
20 IF END#6*K3 THEN 110
```

The IF END# statement is programmable only; it cannot be executed from the keyboard.

In the previous program (page 3-14), ERROR 99 is displayed after the completion of the program, telling you that a PEOF marker is encountered and no more data can be read. This error message can be avoided by including an IF END# statement in the program.

Upon detecting an end of file condition, the program branches to the line number specified in the IF END# statement.† This condition remains in effect until another IF END# statement, with a different line number parameter for the same file, is executed. All previous IF END# conditions are cancelled when a FILES statement is executed, while an ASSIGN statement cancels the IF END# condition only for the individual file specified in the ASSIGN statement.

Consider the previous program (page 3-14) modified to include an IF END# statement.

```
10 FILES DATA15
20 READ #1,14
30 IF END#1 THEN 70
40 READ #1;A,B,C,D,E,F,G,H
50 PRINT A;B;C;D;E;F;G;H
60 GOTO 40
70 PRINT
80 PRINT "END OF DATA HAS BEEN REACHED."
90 END
```

In this program, when all the data is read, the pointer comes to a PEOF marker. The IF END# statement (line 30) sets up a condition whereby the program branches to line 70 when the READ# statement (line 40) encounters the PEOF marker. At this point, lines 70 and 80 are executed, informing you that the PEOF marker is the next item in the file.

† If the line number to which the IF END# statement refers does not exist, ERROR 44 is displayed. This error refers to a PRINT# or READ# statement, not the IF END# statement, because it is actually the PRINT# or READ# which precipitates the error. IF END# simply establishes a condition.

Notice that the IF END# statement is executed only once before entering the READ#/PRINT loop. Since IF END# establishes the exit procedure for this loop, it has to be executed before entering the loop, but should *not* be included in the loop. Repeated, unnecessary execution of IF END# should be avoided because of the additional time needed to execute this statement.

NOTE

An IF END# statement sets up a condition to detect an EOF marker. If you attempt to access a non-existent or invalid record without a previously executed IF END# statement, ERROR 99 is displayed. If a file has not been assigned into an * position which is referenced, or no FILES statement is given, executing the IF END# statement results in ERROR 94.

TYP FUNCTION

The TYP function is used to identify the type of the next item in a specified file.

Syntax:

TYP file number

TYP (–file number)

Examples:

```
   F1=TYPF2
20 IF TYPG↑2-1=3 THEN 50
30 IF TYP(-2)=4 THEN 250
40 GOTO TYPX OF 100,200,300,400,500,600
```

In some cases, the type of data item next on the file may be unknown. Use the TYP function to find out what that data item is. The TYP function returns a number code which can then be used for various purposes. The number codes and their meanings are listed below.

1 – Next item is a full precision number
2 – Next item is a character string
3 – Next item is an end of file marker
4 – Next item is an end of *record* marker
5 – Next item is a split precision number
6 – Next item is an integer precision number

Although serial PRINT# and READ# statements can detect end of file (PEOF and LEOF) markers when an IF END# condition is established, there is no way to detect end of *record* (LEOR or PEOR) markers using the IF END# statement.

The optional minus sign in the TYP syntax provides the only means to check for end of record markers in serial file access mode. Use the minus sign before the file number parameter for this most general case. If a value of 4 is returned, the next item in a record is an LEOR or PEOR marker.

SERIAL FILE ACCESS

(Continued)

Execute the program shown below to store different types of data on the first record of the 5–record file, NU?

```
10 OPEN "NU?",5
20 FILES NU?
30 DIM A[1],B$[5],CS[1],DI[1]
40 A[1]=1111
50 B$="DATA"
60 C[1]=2222
70 D[1]=3333
80 PRINT #1;A[1],B$,C[1],D[1]
90 END
```

Example B

The following 'DATA CHECK' program checks the type of the next data item before reading the item.

```
10 FILES NU?
20 DIM A[50],B$[50],CS[50],DI[50]
30 READ #1,1
40 GOTO TYP(-1) OF 50,90,130,160,190,230
50 REM  THE NEXT ITEM IS A FULL PRECISION NUMBER
60 READ #1;A[1]
70 PRINT A[1]
80 GOTO 40
90 REM  THE NEXT ITEM IS A CHARACTER STRING
100 READ #1;B$
110 PRINT B$
120 GOTO 40
130 REM  THE NEXT ITEM IS AN END OF FILE MARKER
140 PRINT "AN END OF FILE MARKER HAS BEEN DETECTED."
150 GOTO 270
160 REM  THE NEXT ITEM IS AN END OF RECORD MARKER
170 PRINT "AN END OF RECORD MARKER HAS BEEN DETECTED."
180 GOTO 270
190 REM  THE NEXT ITEM IS A SPLIT PRECISION NUMBER
200 READ #1;C[1]
210 PRINT C[1]
220 GOTO 40
230 REM  THE NEXT ITEM IS AN INTEGER PRECISION NUMBER
240 READ #1;D[1]
250 PRINT D[1]
260 GOTO 40
270 END
```

The GOTO...OF statement (line 40) branches the program to one of six line numbers, depending on the value of TYP (–1). Notice that this statement must be executed before each READ# statement, to determine what type of data item is next on the record. The printout of this program is as follows:

```
 1111
DATA
 2222
 3333
AN END OF RECORD MARKER HAS BEEN DETECTED.
```

If the pointer is set to another record of NU? in line 30 (e.g., READ#1,4), when this program is run, TYP (−1) returns a value of 3, indicating an LEOF marker is encountered. This marker was placed in the first word of each record automatically when the file was opened. The LEOF marker disappears only when data is stored in the record.

Example C

To illustrate the principles involved in serial file access, consider the 'ADD DATA' program shown below.

```
10 REM  THIS PROGRAM STORES DATA ITEMS SEQUENTIALLY IN A FILE.
20 FILES *
30 DIM A$[6]
40 DISP "WHAT FILE NAME (1-6 CHARS)";
50 INPUT A$
60 ASSIGN A$,1,X
70 REM  THE NEXT LINE TESTS THE RETURN VARIABLE (X). IF THE FILE DOES NOT
80 REM  EXIST (I.E., X#0), THEN IT IS OPENED.
90 IF X=0 THEN 150
100 DISP "HOW MANY RECORDS IN ";A$;
110 INPUT A
120 OPEN A$,A
130 ASSIGN A$,1,X
140 GOTO 280
150 DISP "ERASE OLD DATA? (YES OR NO)";
160 INPUT A$
170 IF POS(A$,"Y")=0 THEN 210
180 REM  THE NEXT LINE EFFECTIVELY ERASES THE DATA IN THIS FILE BY
190 REM  PLACING AN END OF FILE (LEOF) MARKER IN THE BEGINNING OF THE FILE.
200 PRINT #1;END
210 IF END#1 THEN 280
220 REM  THE NEXT TWO STATEMENTS REPOSITION THE POINTER,
230 REM  DATA ELEMENT BY DATA ELEMENT. WHEN THE END OF FILE (EOF) CONDITION
240 REM  IS DETECTED, THE PROGRAM BRANCHES TO THE LINE NUMBER (LINE 280)
250 REM  SPECIFIED IN THE PREVIOUS IF END# STATEMENT (LINE 210).
260 READ #1;A
270 GOTO 260
280 DISP "WHAT NUMBER SIGNALS END OF DATA";
290 INPUT Q
300 IF END#1 THEN 410
310 I=0
320 DISP "INPUT DATA ELEMENT";
330 INPUT A
340 IF A=Q THEN 380
350 PRINT #1;A,END
360 I=I+1
370 GOTO 320
380 PRINT "DATA ENTRY COMPLETED. ";I;" ITEMS PRINTED IN FILE."
390 DISP "DONE";
400 END
410 PRINT "THERE IS NO MORE ROOM IN THIS FILE.";I;"ITEMS WERE PRINTED."
420 DISP "DONE";
430 END
```

The remarks beginning on lines 10, 70, 180 and 220 explain the separate parts of this program. If the file you want already contains data, and if you want to add your new data to the file, lines 260 and 270 reposition the pointer to the end of the old data. The IF END# statement (line 210) provides an exit procedure for the program. The program branches to line 280 when it encounters an LEOR, PEOR or PEOF marker which was previously placed in the file, or when it encounters the LEOF marker placed by line 200 when all data is effectively erased. In either case, line 280 is eventually executed.

SERIAL FILE ACCESS

(Continued)

Example D

The 'STATISTICS' program shown below reads data in an existing file and calculates the number of items read, the mean, standard deviation, largest and smallest values of the data. Line 180 reads the data, element by element, until the end of record or end of file condition, set up by line 130, is met.

```
10 REM  THIS PROGRAM CALCULATES THE MEAN, STANDARD DEVIATION,
20 REM  AND LARGEST AND SMALLEST VALUES IN A GIVEN DATA FILE.
30 FILES *
40 DIM A$[6]
50 DISP "WHAT FILE NAME (1-6 CHARS)";
60 INPUT A$
70 ASSIGN A$,1,X
80 IF X=0 THEN 130
90 PRINT A$;" DOES NOT EXIST."
100 PRINT
110 DISP "DONE";
120 END
130 IF END#1 THEN 270
140 S1=S2=0
150 M1=9.99999999999E+99
160 M2=-M1
170 N=1
180 READ #1;A
190 S1=S2+A
200 S2=S2+A↑2
210 IF A>M1 THEN 230
220 M1=A
230 IF A<M2 THEN 250
240 M2=A
250 N=N+1
260 GOTO 180
270 N=N-1
280 PRINT "NUMBER OF VALUES =";N
290 PRINT "MEAN: ";S1/(N+(N=0))
300 IF N<3 OR S1=S2 THEN 320
310 PRINT "STANDARD DEVIATION: ";SQR((N*S2-S1↑2)/(N*(N-1)))
320 PRINT "LARGEST VALUE: ";M2;"SMALLEST VALUE: ";M1
330 PRINT
340 DISP "DONE";
350 END
```

RANDOM FILE ACCESS

Data stored in a random manner is stored into specified physical records within a file. Variations of the previously discussed PRINT# and READ# statement syntaxes are used to access data in particular records. As in serial file access, a pointer keeps track of the data item currently being accessed. Unlike serial file access, however, in random file access, a specific record number within a file must be specified in each PRINT# and READ# statement. Consequently, the pointer is positioned at the beginning of the specified record before printing or reading operations occur. Data is printed or read consecutively from the beginning of the physical record.

RANDOM PRINT# STATEMENT

The random PRINT# statement prints data in the form of variables, numbers or strings of characters from the beginning of a specified physical record. A variation of this syntax can also be used to erase the contents of an individual record.

Syntax:

[line number] PRINT# file number, record number; list [,END]

[line number] PRINT# file number, record number [;END]

Examples:

```
10 PRINT #3,2;A,B,C$
20 PRINT #1,K/2+1;X,Y,Z
30 PRINT #2,1;X,X$,END
   PRINT #4,7;END
   PRINT #3,1;X/4,"VOLTAGE",3.021
```

The parameters in this syntax are similar to the serial PRINT# statement parameters in usage and restrictions. The pointer is positioned at the beginning of the specified record *before* the PRINT# statement is executed.

The program below prints consecutive numbers onto each odd–numbered physical record of a 10–record file named, TEN.

```
10 OPEN "TEN",10
20 R=A=1
30 FILES TEN
40 IF R>10 THEN 90
50 PRINT #1,R;A
60 A=A+1
70 R=R+2
80 GOTO 40
90 END
```

In line 50, the record number parameter is specified by the variable, R. Line 70 increments this variable by 2 so that only odd–numbered records are accessed.

RANDOM FILE ACCESS

(Continued)

By printing in specific records of the file TEN, previous data in those records is erased and replaced by the new data. An LEOR marker is automatically placed at the end of each data list (i.e., the one data item, A) in each odd—numbered record.

File TEN now contains the following information.

Record #	Data
1	1
2	—
3	2
4	—
5	3
6	—
7	4
8	—
9	5
10	—

When no list is specified in a PRINT# statement, the following syntax erases the contents of a particular record.

[line number] PRINT# file number, record number

This syntax actually places an LEOR marker at the beginning of the specified record, making the data contained in the physical record inaccessible to any READ# statement. When an LEOR marker is detected, a random READ# statement (see next section) encounters an end of record condition; a serial READ# statement skips over the entire record and attempts to access data in the next record. Of course, a subsequent PRINT# statement writes over the LEOR marker.

The following program erases every third record of file, TEN, which was opened and accessed in the previous program.

```
10 A=1
20 FILES TEN
30 PRINT #1,A
40 A=A+3
50 IF A>10 THEN 70
60 GOTO 30
70 END
```

The information which is now left in the file is shown below.

Record #	Data
1	–
2	–
3	2
4	–
5	3
6	–
7	–
8	–
9	5
10	–

An LEOF marker is stored by executing a statement with the following syntax.

[line number] PRINT# file number, record number; END

This places an LEOF marker in the first word of a specified record in the same way that an LEOF marker is placed automatically when the file is originally opened. If a random READ# or a serial READ# statement attempts to access a record with an LEOF marker at the beginning of that record, an end of file condition is established.

By including a list of variables before the END parameter, an LEOF marker is stored after the data within the physical record specified, in place of the LEOR marker usually written by the mass memory system. If the serial or random READ# list is greater than the PRINT# list which stored the data, the LEOF marker is detected, establishing an end of file condition.

RANDOM READ# STATEMENT

The random READ# statement reads numbers and strings into variables from a specified record in a file, starting from the beginning of that record. Substrings are not allowed. A variation of this syntax can be used to reposition the pointer (see page 3-13).

Syntax:

[line number] READ# file number, record number [;list]

Examples:

```
10 READ #3,4;A,B,C$
20 READ #4,2*J
   READ #M,N;A,B1,B2
```

As in the case of serial READ# statements, the variables into which you read data items do not necessarily have to be the same variables from which you printed the data items on the record.

RANDOM FILE ACCESS

(Continued)

The data item list is omitted only when you are repositioning the pointer as discussed on page 3-13. The pointer is positioned at the beginning of the specified record *before* the READ# statement is executed.

The following program reads the data printed in the 5th and 9th records of the file, TEN.

```
10 FILES TEN
20 READ #1,5;X
30 READ #1,9;Y
40 PRINT "THE DATA ITEM IN RECORD #5 IS EQUAL TO";X
50 PRINT "THE DATA ITEM IN RECORD #9 IS EQUAL TO";Y
70 END
```

This data was originally printed onto odd–numbered records of file TEN (page 3-21). The data on records 1 and 7 was erased. The program above reads the data from records 5 and 9 and then prints it on the calculator printer. If the calculator is programmed to read data from each record, ERROR 99 is displayed, indicating that an LEOR marker is detected at the beginning of record 1.

The printout from this program is shown below.

```
THE DATA ITEM IN RECORD #5 IS EQUAL TO 3
THE DATA ITEM IN RECORD #9 IS EQUAL TO 5
```

IF END# STATEMENT

The IF END# statement sets up a condition in the program. If any end of record marker or end of file marker is encountered during the execution of a random READ# statement, or a PEOR or PEOF marker is encountered during the execution of a random PRINT# statement, the program branches to the line number specified in the previous IF END# statement, thus avoiding an ERROR.

Syntax:

line number IF END# file number THEN line number

Examples:

```
10 IF END#3 THEN 222
20 IF END#6*K/3 THEN 110
```

The IF END# statement is programmable only; it cannot be executed from the keyboard. As mentioned earlier in serial file access, the IF END# statement sets up a condition under which a PRINT# or READ# statement branches to the specified line† when an end of file marker is encountered. In random file access, however, the IF END# condition can be used to check for either an EOF marker or an *EOR* marker. If, for example, a random READ# data list includes more data items than were printed by the previous PRINT# statement, the end of record condition is detected.

† If the line number to which the IF END# statement refers does not exist, ERROR 44 is displayed. This error refers to a PRINT# or READ# statement, not the IF END# statement, because it is actually the PRINT# or READ# which precipitates the error. IF END# simply establishes a condition.

Consider the following program.

```
10 OPEN "POWER",10
20 FILES POWER
30 A=1
40 IF END#1 THEN 80
50 PRINT #1,A;A,A↑2,A↑3,A↑4
60 A=A+1
70 GOTO 50
80 PRINT
90 PRINT "NO MORE ROOM IN THIS FILE."
100 END
```

This program prints four data items onto each record of the file, POWER. When variable A is incremented to a value greater than 10 – the number of records in the file – the condition set up by the IF END# statement in line 40 is met. The program branches to line 80 when the PEOF marker is encountered in this case. In this way, ERROR 99 is avoided.

Similarly, the IF END# statement can be used to avoid an error message when *reading* data from a file, as shown below.

```
10 FILES POWER
20 A=1
30 IF END#1 THEN 80
40 READ #1,A;W,X,Y,Z
50 PRINT W,X,Y,Z
60 A=A+1
70 GOTO 40
80 PRINT
90 PRINT "THE END OF THIS FILE HAS BEEN REACHED."
100 END
```

The printout for the above program is shown below.

```
 1             1             1             1
 2             4             8             16
 3             9             27            81
 4             16            64            256
 5             25            125           625
 6             36            216           1296
 7             49            343           2401
 8             64            512           4096
 9             81            729           6561
 10            100           1000          10000

THE END OF THIS FILE HAS BEEN REACHED.
```

RANDOM FILE ACCESS

(Continued)

IF END# can also be used to branch to a specified program line when there is a possibility of printing more data onto a physical *record* than it can hold. In this case, IF END# sets up an exit procedure for the PRINT# statement when it encounters a PEOR marker.

NOTE

An IF END# statement sets up a condition to detect an EOF or an EOR marker. If you attempt to access a non-existent or invalid record without a previously executed IF END# statement, ERROR 99 is displayed. If a file has not been assigned into an * position which is referenced, or no FILES statement is given, executing the IF END# statement results in ERROR 94.

TYP FUNCTION

The TYP function is used to identify the type of the next item in a specified file.

Syntax:

TYP file number

TYP (–file number)

Examples:

```
   F1=TYPF2
20 IF TYPGT2-1=3 THEN 50
30 IF TYP(-2)=4 THEN 250
40 GOTO TYPX OF 100,200,300,400,500,600
```

In some cases, the type of data item next on the file may be unknown. Use the TYP function to find out what that data item is. As discussed earlier in serial file access, the TYP function returns a number code which can then be used for various purposes. The number codes and their meanings are listed below.

1 – Next item is a full precision number
2 – Next item is a character string
3 – Next item is an end of file marker
4 – Next item is an end of record marker
5 – Next item is a split precision number
6 – Next item is an integer precision number

The TYP function provides an alternate method of determining whether an EOR or EOF marker will be encountered during a random PRINT# or random READ# operation. While an IF END# statement sets up a condition to check for either of these markers, TYP can identify the specific marker encountered. This information can be used in a GOTO...OF statement to branch to different parts of the program depending on the value returned for the function. As mentioned earlier, the minus sign must be used before the file number parameter to detect an EOR marker.

See "TYP Function" (page 3-17) for an example of TYP function use.

NOTES

Chapter 4

SUPPLEMENTARY COMMANDS

This chapter discusses matrix operations, the use of more than one platter in a system, and various other commands for increased control of your mass memory.

MATRIX OPERATIONS

In order to work with matrices and your mass memory system, you need an 11270 Matrix Operations ROM installed in your calculator. Then, with the MAT READ# and MAT PRINT# commands, you can print or read entire matrices onto a data file. As with the statements and commands discussed in Chapter 3, particular attention should be paid to matrix command syntaxes. The '#' character is included in the matrix commands to differentiate them from the BASIC matrix commands described in the Matrix ROM Operating Manual.

MAT PRINT# STATEMENT

The MAT PRINT# statement prints an entire matrix onto a specified record or file.

Syntax:

[line number] MAT PRINT# file number [,record number] ; list of matrix variables

Examples:

```
10 MAT PRINT # 2;A
20 MAT PRINT # 1,6;X
   MAT PRINT # 9*J/3,K↑2;X,Y,Z
```

Matrices can only be printed on data files; the size of the matrix you want to store is limited by the number of records you specify when opening that file. For example, a one–record file, which contains 256 words, can hold up to 64 full precision numbers. This means that an 8 by 8 matrix is the largest matrix of full precision numbers that file (or one record of any other file) can hold. Of course, by printing a matrix serially (i.e., not specifying a particular record), the matrix size is not limited to 256 words. In this case it is limited by the number of records in the file multiplied by 256 words per record.

MATRIX OPERATIONS

(Continued)

The following example prints a 4 by 4 matrix serially into the file, MATRIX.

```
10 OPEN "MATRIX",5
20 DIM A[4,4]
30 FILES MATRIX
40 FOR I=1 TO 4
50 FOR J=1 TO 4
60 A[I,J]=10*I+J
70 NEXT J
80 NEXT I
90 MAT PRINT # 1;A
100 END
```

The elements of a matrix are printed consecutively in row–column order from the beginning of the file or record specified. With the record number parameter omitted, the matrix is printed onto the file from the position of the pointer. By including this optional parameter, the matrix is printed onto a single specified record.

If the matrix is too large for the file or record specified, an EOR or EOF marker is encountered and ERROR 99 is displayed. Of course, an IF END# statement can be used to detect the EOR or EOF condition and avoid this error message.

MAT READ# STATEMENT

The MAT READ# statement reads the matrix from a specified record or file.

Syntax:

[line number] MAT READ# file number [,record number]; list of matrix variables

Examples:

```
10 MAT READ # 2;A
20 MAT READ # 1,7;X
30 MAT READ # 3,4;Y[5,4]
   MAT READ # X↑3,J/2;K
```

To read the matrix created in the previous section and print it on the calculator printer, use the following program.

```
10 DIM B[4,4]
20 FILES MATRIX
30 MAT READ # 1;B
40 MAT PRINT B
50 END
```

Line 30 reads the matrix from MATRIX into the calculator, using the mass memory MAT READ# statement. Line 40 then prints the entire matrix, using the Matrix ROM MAT PRINT statement. Notice that although matrix variable A was used to define the original matrix, any variable can be used to read it back again (in this case, variable B is used). This matrix must be dimensioned, as in line 10. The printout from this program is shown below.

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

The matrix can be read in any format less than or equal to the original size. In the program below, a 4 by 4 matrix is read as an 8 by 2 matrix by dimensioning the matrix variable B in that manner.

```
10 DIM B[8,2]
20 FILES MATRIX
30 MAT READ # 1;B
40 MAT PRINT B
50 END
```

The printout follows.

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

In the above printout, data element (2,1) is 13. This is because the data elements of the original matrix were printed on the file point by point; they were not stored in unique random locations.

MATRIX OPERATIONS

(Continued)

If you run a program in which a matrix was dimensioned larger than the original matrix, ERROR 99 is displayed. This error message can be avoided, however, by using an IF END# statement to detect the end of record condition.

An implicit REDIM statement can be included in the MAT READ# statement. To read the data in this matrix in a 3 by 4 format, use this program.

```
10 DIM C[4,4]
20 FILES MATRIX
30 MAT READ # 1;C[3,4]
40 MAT PRINT C
50 END
```

Notice that the DIM statement (line 10) dimensions a matrix that is at least as large as that specified by the implicit REDIM statement in line 30. The printout of this program is as follows.

```
 11          12          13          14

 21          22          23          24

 31          32          33          34
```

MULTIPLE PLATTERS

As you know, a mass memory system can contain from one to four platters. You must specify which platter you want to address, since only one platter can be accessed at a time. Once this has been specified, the platter is controlled with the commands and statements previously discussed. All control, program and data commands apply to the platter designated.

The 9867B Mass Memory Drive, which contains two platters, must be set to one of the dual positions:

0 and 1 or 2 and 3

The 9867A Mass Memory Drive can be switched to any of the four positions, provided that no other unit is switched to the same position. See page 1-9 for instructions on setting the unit select switch.

UNIT COMMAND

The UNIT command specifies the platter to be used for subsequent commands.

Syntax:

[line number] UNIT unit number

Examples:

```
10 UNIT 2
   UNIT X↑2-1
```

Once the UNIT command is executed, all subsequent statements and commands reference the platter specified until a new UNIT command, SCRATCH A, or LOAD BIN is executed or the calculator is switched OFF. Under these conditions, a previous FILES statement is also destroyed. When the calculator is first switched ON, or after SCRATCH A or LOAD BIN is executed, UNIT 0 is automatically specified.

Assume that you have the following program stored in a program file named, PRINT0.

```
10 FOR I=1 TO 5
20 PRINT I
30 NEXT I
40 END
```

To transfer this program from the file, PRINT0 on platter 0 to a file (call it PRINT2) on platter 2, execute the following steps from the keyboard:

```
GET "PRINT0"
UNIT 2
SAVE "PRINT2"
```

Notice that platter 0 does not have to be specified originally, since it is automatically specified when your calculator is turned ON.

Now assume you have three data items stored on the fifth record of the file, GIVE on platter 2. To copy this data from that file to the sixth record of a new file named, TAKE on platter 3, execute the following program:

```
10 UNIT 2
20 FILES GIVE
30 READ #1,5;A,B,C
40 UNIT 3
50 OPEN "TAKE",10
60 FILES TAKE
70 PRINT #1,6;A,B,C
80 END
```

Notice that a new FILES statement (line 60) must be executed so that subsequent statements refer to the correct (new) file.

MULTIPLE PLATTERS

(Continued)

The UNIT command obsoletes all previous FILES statements. If the UNIT command references an unconnected platter, no error occurs, but subsequent commands result in error messages.

PLATTER-DUPLICATE PROCEDURE

All of the information contained on one platter can be copied to another platter† by performing the PLATTER-DUPLICATE procedure.

It should be emphasized that all of the information contained on the first platter – including data, programs, EOR and EOF markers, and unused space – are duplicated on the second platter, erasing everything on the latter platter.

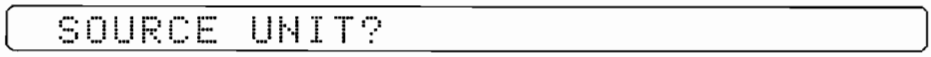
The second platter, as well as the first, must be initialized before the PLATTER-DUPLICATE procedure is performed.

Insert the mass memory system tape cassette in the calculator and execute:

LOAD BIN 1

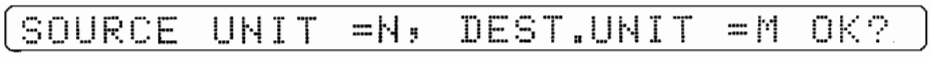
After the display returns, perform this procedure:

1. Key in PLATTER-DUPLICATE and press EXECUTE.
   The display will show:



2. Key in (but don't execute) the unit number of the original platter.
   The display will then show:

3. Key in (but don't execute) the number of the second platter.
   The display will show:



4. Carefully verify that you keyed in the correct numbers for N and M. If so, press EXECUTE. If the numbers are *not* correct, press STOP, CLEAR, and begin the procedure again from step 1.

The platter is duplicated automatically at this point. Duplication takes about 3 minutes. ERROR 903, if displayed, indicates duplication is not allowed. See footnote.

† In general, if the second, or 'destination' platter contains defective tracks, the DUPLICATE-PLATTER procedure is not performed. Defective tracks on the first, or 'source' platter do not affect the procedure. If the defective tracks on the destination platter happen to coincide with those on the source platter, the procedure can be executed, also.

MISCELLANEOUS COMMANDS

The following commands provide greater flexibility for controlling your mass memory system.

DCOPY	DFDUMP
DFLOAD	DREN
DGET	DBYTE
DEXP	

A string variable must be used in the DBYTE and DEXP commands; strings are not permitted in any of the other commands listed above.

DCOPY COMMAND

The DCOPY command duplicates the contents of one data file onto another. These two data files can be located on separate platters.

Syntax:

```
[line number] DCOPY "1st file name" [,unit number]
              TO "2nd file name" [,unit number]
```

Examples:

```
10 DCOPY "PERSON" TO "PEOPLE"
   DCOPY "MAX",1 TO "MIN",3
```

The optional unit number parameters need not be used if both the first file and the second are located on the same platter. In any event, DCOPY does not change the unit number reference that existed previously in the calculator memory.

The DCOPY command can be used to duplicate data onto a file as long as that file is large enough to accommodate it. This command copies data only, not available space. For example, a five-record file can be copied into a two-record file provided that no more than two of the source file's five records are filled with data.

ERROR 96 results if the second file is smaller than the current size of the first file.

The second file must be opened before executing the DCOPY command. Also, both files must either have identical protection codes, or none at all. Remember that only a data file can be copied with the DCOPY command; a program or key file cannot.

MISCELLANEOUS COMMANDS

(Continued)

DFDUMP COMMAND

The DFDUMP (data file dump) command stores a specified data file, presently on the platter, onto the calculator internal cassette(s).

Syntax:

DFDUMP "file name"

Example:

```
DFDUMP"TRUCK"
```

The DFDUMP command stores a platter data file of any size onto one or more tape cassettes. The data file can be protected or unprotected; no protection code parameter is required.

Each cassette can hold up to a 150–record file; larger files are stored on more than one tape cassette according to the procedure explained below.

The DFDUMP command is executed from the keyboard only, because calculator memory is erased when this command is executed.

If the file you wish to store is larger than 150 records, more than one tape cassette is necessary. Insert the first tape cassette in the tape transport, close the transport door and press REWIND. When the tape is on clear-leader, key in and execute the DFDUMP command from the keyboard, specifying the file you wish to store on the cassettes. The first 150 records of the file are stored, one by one, onto the first tape cassette. When the storing is completed, the tape cassette is rewound to clear-leader automatically and the following message is displayed†:

```
LOAD ANOTHER TAPE
```

Remove the first cassette and mark it so that the order of cassettes remains the same when you want to load the information on another mass memory file. Insert the next cassette in the tape transport, close the transport door and press REWIND. When the tape is on clear-leader, press EXECUTE to continue storing the file, from record number 151. Follow this procedure until the entire file is stored on your cassettes. The last cassette required is rewound and the display shows:

```
├
```

† If the file to be stored is less than 150 records, the same procedure must be followed. No message is displayed, but calculator control returns after the tape is completely rewound.

The following facts should be remembered when using the DFDUMP command.

- The calculator memory is erased when DFDUMP is executed, so execute this command only from the keyboard.
- The information (e.g., BOF control markers, data) currently on the cassette(s) in the space required for storage is replaced by the data stored from the platter.
- The tape cassette(s) must be positioned on clear-leader before DFDUMP is executed.
- Only consecutive, non-empty records and the first empty record of a file is stored by the DFDUMP command. See below.
- Once the data from the platter has been stored on the cassette(s), it cannot be randomly accessed from the cassette(s); it can be loaded only onto another mass memory platter with the DFLOAD command, discussed in the next section.

If every physical record in a file contains data, the entire file is stored onto the cassette(s) with the DFDUMP command. If, however, only some of the records of a file contain data, every consecutive data record, up to and including the first empty one in which an LEOF marker is placed, is stored on the cassette(s).

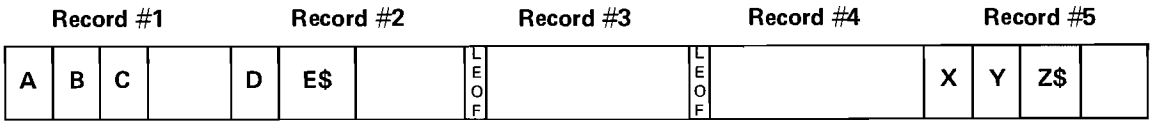


Figure 4—1. A Five—Record File

Consider the 5-record file above. Records 1, 2 and 5 contain data, but records 3 and 4 are empty. When the DFDUMP command is executed, records 1 and 2 are stored, one at a time. Then the mass memory system encounters the LEOF marker which was placed in the first word of the third record when the file was opened. This first empty record (record 3) is also stored on the cassette.

NOTE

Every consecutive data record, up to and including the first empty one, is stored on the cassette(s), regardless of whether the last record that contains data includes an EOR or an EOF marker.

In the sample file above, the data in record 5 is not stored on the cassette because at least one empty record (in this case, two) separates the non-empty data records. The LEOF marker in record 3 signals the mass memory system to store the entire empty record and immediately rewind the tape cassette. The DFDUMP command is completed at that point.

In the case of a completely empty file, the first record of the file is stored, since it is the first empty record and contains an LEOF marker in its first word.

MISCELLANEOUS COMMANDS

(Continued)

DFLOAD COMMAND

The DFLOAD (data file load) command loads the data from the calculator internal tape cassette(s) to a specific file on the platter.

Syntax:

DFLOAD "file name"

Example:

```
DFLOAD"UP"
```

The DFLOAD command loads the data which is on one or more tape cassettes onto a platter data file. If the data on the cassette(s) is protected (i.e., originally stored from a protected data file), DFLOAD loads it onto a protected data file as long as its protection code matches the original file's code; the data on an unprotected cassette (i.e., originally stored from an unprotected data file) can be loaded only onto an unprotected data file.

Like DFDUMP, the DFLOAD command is executed from the keyboard only, because calculator memory is erased when this command is executed.

If more than one cassette is to be loaded onto a platter data file (i.e., if more than 150 records are required), insert the first tape cassette in the tape transport, close the transport door and press REWIND. When the tape is on clear-leader, key in and execute the DFLOAD command from the keyboard, specifying the file in which you wish to store the cassettes' data. The first 150 records are loaded, one by one. When the loading is completed, the tape cassette is rewound to clear-leader automatically and the following message is displayed†:

```
LOAD ANOTHER TAPE
```

Remove the first cassette and insert the next cassette in the tape transport. Close the transport door and press REWIND. When the tape is on clear-leader, press EXECUTE to continue loading the file, from record 151. Follow this procedure until the entire file is loaded from your cassettes. The last cassette is rewound and the display shows:

```
├
```

† If the file to be loaded is less than 150 records, the same procedure must be followed. No message is displayed, but calculator control returns after the tape is completely rewound.

DREN COMMAND

The DREN (rename) command allows you to change the name of any file.

Syntax:

[line number] DREN "old file name" TO "new file name" ["protection code"]

Examples:

```
10 DREN "DAD" TO "SON"
   DREN "LIL" TO "LILLY", "WIFE"
```

The DREN command renames your file. The contents of the file remain the same. A protection code, if present, must be specified. This protection code is transferred to the new file name automatically. The old file name no longer exists on the platter.

DGET COMMAND

The DGET command loads the source (non-compiled) program specified into the calculator and checks it for syntax errors.

Syntax:

[line number] DGET "file name" [0]

Examples:

```
10 DGET "NATHAN"
   DGET "NAT" 0
```

A source program is a series of statements which are printed sequentially onto a data file as a series of string variables, one BASIC statement per string. After the last statement (string), an LEOF marker should be written by executing a PRINT#...END statement to separate old data from the new, source program. Using DGET, you can load these statements into the calculator and perform BASIC syntax checks at the same time. The program is executed immediately unless the optional parameter, 0, is included in the syntax. In this case, the program is loaded into calculator memory, but not executed.

The DGET command provides a means to use programs actually written by other programs (see Example G, page 4-14).

MISCELLANEOUS COMMANDS

(Continued)

DBYTE COMMAND

The DBYTE command converts the value of a specified variable to its binary equivalent character. This binary character is then stored as a single character in the specified string.

Syntax:

[line number] DBYTE variable, string name

Examples:

```
10 DBYTE X,D$
   DBYTE A,R$
```

Obviously, the variable must be the decimal equivalent of an ASCII character. For example, if you set Y equal to 34, A$ will contain the quotation character (") when you execute the following syntax:

```
DBYTE Y,A$
```

Only simple variables (no expressions) and simple strings (no subscripts) are allowed.

Example E

The 'CHARACTER' program below generates the Model 9866A Printer characters associated with their decimal codes, from 1 to 126. The numbers less than 33 and greater than 95 are not needed for 9866A printer operation. For the characters and functions these codes represent on the 9861A Typewriter and the 38 ASR Teleprinter, see Table F-2 of the Model 30 Operating and Programming Manual.

```
10 DIM A$[1]
20 FOR I=1 TO 126
30 DBYTEI,A$
40 PRINT I,A$
50 NEXT I
60 END
```

DEXP COMMAND

The DEXP command converts the value of the specified variable into a 4-digit character string with leading zeros.

Syntax:

[line number] DEXP variable, string name

Examples:

```
10 DEXP Z,A$
   DEXP M1,P$
```

The DEXP command can be used, for example, to generate line numbers for BASIC statements. If you set X equal to 10, A$ will contain the 4-digit character string, 0010 by executing the following command.

```
   DEXP X,A$
```

Example F

Shown below is a 'CHECK-WRITING' program that prints amounts (in dollars and cents) with leading asterisks. This is useful when writing checks. This program converts a dollar value (less than $10,000, but not negative) to a string with leading asterisks.

```
10 DIM A$[15],B$[5]
20 A$="**********.**"
30 DISP "DOLLAR VALUE";
40 INPUT N
50 IF N >= 10000 OR N<0 THEN 30
60 N1=100*(N-INTN)
70 IF N1#0 THEN 100
80 B$="0000"
90 GOTO 110
100 DEXP N1,B$
110 A$[14]=B$[3]
120 N1=INTN
130 IF N1#0 THEN 160
140 B$="0"
150 GOTO 200
160 DEXP N1,B$
170 IF B$[1,1]#"0" THEN 200
180 B$=B$[2]
190 GOTO 170
200 B$[2]=B$
210 B$[1,1]="$"
220 A$[13-LEN(B$),12]=B$
230 PRINT A$
240 PRINT
250 GOTO 20
260 END
```

MISCELLANEOUS COMMANDS

(Continued)

Example G

The program on the facing page illustrates one way in which the DREN, DGET, DBYTE and DEXP commands can be used. This program generates the 'source' program, overprinted on the listing in solid black. The new source program is used to rename an existing data file.

The FILES statement in line 40 is referenced by ASSIGN statements in lines 50, 120 and 160. Line 190 sets the variable, Z, to 34. The subsequent DBYTE command (line 200) stores the equivalent printer character (") in D$. Variable L is set to 10, 20 and 30 as the program progresses. These values are used by the DEXP command (lines 220, 330 and 400) to create 4-digit character strings which form line numbers in the source program.

Lines 150 to 420 actually generate and store the 3-line source program on the data file, /////. The DGET command (line 430) loads and runs this source program.

Assume this program was run to change the name of a data file from DARK to LIGHT. The calculator memory, which now contains the source program, is listed below.

```
10 DREN"DARK"TO"LIGHT"
20 PRINT "DONE"
30 END
```

The first statement in the source program uses the DREN command to change the file name. Although the DREN command cannot include string variables for the file name parameters, these names are generated using the string variable, C$, which is printed, as data, onto the data file, /////.

```
10 DIM A$[6],B$[6],C$[80],D$[1],E$[5]
20 DISP "PRESENT FILE NAME (1-6 CHARS.)";
30 INPUT A$
40 FILES *,*,*
50 ASSIGN A$,1,J
60 IF J=3 THEN 440
70 DISP " IS FILE PROTECTED?  1=YES; 0=NO";
80 INPUT X
90 IF X=1 THEN 500
100 DISP "NEW FILE NAME (1-6 CHARACTERS)";
110 INPUT B$
120 ASSIGN B$,2,K
130 IF K=0 THEN 470
140 PRINT "FILE ";A$;" WILL NOW BE CALLED ";B$
150 E$="/////"
160 ASSIGN E$,3,Y
170 IF Y=0 THEN 190
180 OPEN E$,2
190 Z=34
200 DBYTE Z,D$                    }
210 L=10                          }
220 DEXPL,C$                      }
230 C$[LEN(C$)+1]="DREN"          }
240 C$[LEN(C$)+1]=D$              }
250 C$[LEN(C$)+1]=A$              }  10 DREN"A$"TO"B$"
260 C$[LEN(C$)+1]=D$              }
270 C$[LEN(C$)+1]="TO"            }
280 C$[LEN(C$)+1]=D$              }
290 C$[LEN(C$)+1]=B$              }
300 C$[LEN(C$)+1]=D$              }
310 PRINT #3;C$                   }
320 L=20                          }
330 DEXPL,C$                      }
340 C$[LEN(C$)+1]="PRINT"         }
350 C$[LEN(C$)+1]=D$              }  20 PRINT"DONE"
360 C$[LEN(C$)+1]="DONE"          }
370 C$[LEN(C$)+1,80]=D$           }
380 PRINT #3;C$                   }
390 L=30                          }
400 DEXPL,C$                      }  30 END
410 C$[LEN(C$)+1]="END"           }
420 PRINT #3;C$,END               }
430 DGET"/////"
440 PRINT A$;" DOES NOT EXIST.  IS IT SPELLED CORRECTLY?"
450 PRINT
460 GOTO 20
470 PRINT "FILE NAME ";B$;" IS IN USE.  SELECT ANOTHER NAME."
480 PRINT
490 GOTO 100
500 PRINT "PROTECTED FILES CANNOT BE RENAMED USING THIS PROGRAM."
510 PRINT
520 END
```

Source Program

NOTES

Chapter 5
APPLICATIONS

This chapter consists of two applications to give you an idea of the large amounts of data that can be manipulated quickly and easily, using the mass memory.

EXAMPLE 1-DATA BASE PROGRAM

This is called a 'data base' program because, by using it, data can be organized, stored and modified on a data file of the mass memory. In the particular variation shown on the following pages, the program stores employees' names and the number of hours they worked during the five weekdays.

This program is stored on seven Special Function keys (FO through F6). Each of the key functions is discussed separately. A String Variables ROM is required to run this program.

```
10 REM  START (KEY 0)
20 REM INITIALIZES VARIABLES;  A$=FILE NAME"
30 DIM A$[6],B$[6]
40 FILES *
50 DISP "WHAT IS NAME OF YOUR DATA FILE";
60 INPUT A$
70 ASSIGN A$,1,T
80 IF T=0 THEN 110
90 OPEN A$,200
100 GOTO 70
110 DISP "READY";
120 END
```

The program segment loaded on SF key 0, shown above, initializes the two string variables which are used later in this and other program segments. It opens a data file of 200 records, if the file name input has not been previously opened. Finally, it displays the word, READY, when finished.

To run this program segment, press RUN (but don't execute it) and the Special Function key 0. (The RUN key must be pressed before SF key 0 to initialize variables, but RUN should not be executed for subsequent SF keys.)

```
10 REM  CREATE (KEY 1)
20 REM  THIS PROGRAM CREATES THE SPECIFIED FILE AND STORES A 15 CHARACTER"
30 REM  EMPLOYEE FOR EACH RECORD.  LATER THIS RECORD CONTAINS ADDITIONAL DATA.
40 PRINT "TO END RECORD ENTRIES, TYPE DONE."
50 IF END#1 THEN 150
60 PRINT
70 FOR I=1 TO 200
80 DISP "EMPLOYEE";I;"NAME";
90 INPUT A$
100 IF A$="DONE" THEN 150
110 A$[1,15]=A$
120 PRINT #1,I;A$
130 PRINT A$
140 NEXT I
150 DISP "DONE";
160 END
```

The program segment loaded on SF key 1, shown above, allows you to enter up to 200 employees' names on the data file specified in the previous program segment. To run this and all subsequent program segments, simply press the desired SF key.

EXAMPLE 1-DATA BASE PROGRAM

(Continued)

```
10 REM  INPUT (KEY 2)
20 REM  THIS PROGRAM READS THE SPECIFIED FILE, DISPLAYS THE 15 CHARACTER
30 REM  EMPLOYEE AND STORES THE EMPLOYEE AND THE DATA ITEMS IN THE PROPER
40 REM  RECORD.
50 IF END#1 THEN 170
60 FOR I=1 TO 200
70 READ #1,I;A$
80 PRINT A$;"------ ENTER ONE WEEK OF HOURS WORKED, SEPARATED BY COMMAS."
90 PRINT
100 PRINT
110 INPUT A1,A2,A3,A4,A5
120 PRINT #1,I;A$,A1,A2,A3,A4,A5
130 PRINT
140 PRINT A$;A1;A2;A3;A4;A5
150 PRINT
160 NEXT I
170 DISP "DONE"
180 END
```

The program segment loaded on SF key 2, shown above, allows you to enter the number of hours each employee worked in five days.

```
10 REM  PRINT (KEY 3)
20 REM  THIS PROGRAM PRINTS A REPORT OF THE INFORMATION STORED IN THE SPECIFIED"
30 REM  FILE, WITH PROPER HEADINGS.
40 PRINT
50 PRINT
60 PRINT "EMPLOYEE","MON   TUES  WED   THURS  FRI"
70 PRINT
80 IF END#1 THEN 180
90 FOR I=1 TO 200
100 READ #1,I;A$
110 PRINT A$;
120 IF TYP(-1)=4 THEN 160
130 READ #1;A
140 PRINT A;
150 GOTO 120
160 PRINT
170 NEXT I
180 PRINT
190 DISP "DONE"
200 END
```

The program segment loaded on SF key 3, shown above, prints a table of the employees and their hours worked, on the printer.

```
10 REM  SUMMARY (KEY 4)
20 REM  THIS PROGRAM PRINTS SUMMARY REPORTS FOR THE DATA IN THE SPECIFIED
30 REM  FILE.
40 PRINT
50 PRINT
60 T1=T2=T3=T4=T5=T6=T7=0
70 PRINT "EMPLOYEE","HOURS"
80 PRINT
90 IF END#1 THEN 200
100 FOR I=1 TO 200
110 READ #1,I;A$,A1,A2,A3,A4,A5
120 T1=T1+A1
130 T2=T2+A2
140 T3=T3+A3
150 T4=T4+A4
160 T5=T5+A5
170 T9=A1+A2+A3+A4+A5
180 PRINT A$,T9
190 NEXT I
200 PRINT
210 PRINT
220 T0=T1+T2+T3+T4+T5
230 PRINT "TOTALS","MON   TUES  WED  THURS  FRI  WEEK'S TOTAL"
240 PRINT
250 PRINT TAB15,T1;T2;T3;T4;T5;T0
260 PRINT
270 PRINT
280 DISP "DONE"
290 END
```

The program segment loaded on SF key 4, shown above, prints a table of the employees and the *total* number of hours worked per employee and per day.

```
10 REM  MODIFY (KEY 5)
20 REM  THIS PROGRAM ADDS RECORDS AT THE END OF THE EXISTING FILE.
30 IF END#1 THEN 70
40 FOR I=1 TO 200
50 READ #1,I;A$
60 NEXT I
70 PRINT "TO END RECORD ENTRIES, TYPE DONE"
80 PRINT
90 DISP "NAME OF NEW EMPLOYEE";
100 INPUT A$
110 IF A$="DONE" THEN 160
120 DISP "WHAT ARE THE HOURS WORKED";
130 INPUT A1,A2,A3,A4,A5
140 PRINT #1,I;A$,A1,A2,A3,A4,A5
150 GOTO 90
160 DISP "DONE"
170 END
```

The program segment loaded on SF key 5, shown above, allows you to *add* new employees and their hours worked to the existing file. Of course, SF keys 3 and 4 can be pressed anytime to give you an updated table of employees and their hours.

EXAMPLE 1-DATA BASE PROGRAM

(Continued)

```
10 REM  DELETE (KEY 6)
20 REM  THIS PROGRAM DELETES THE SPECIFIED DATA RECORD.
30 DIM B$[15]
40 PRINT "TO END DELETIONS, TYPE DONE."
50 PRINT
60 IF END#1 THEN 70
70 DISP "WHICH EMPLOYEE WILL BE DELETED";
80 INPUT A$
90 IF A$="DONE" THEN 230
100 FOR I=1 TO 200
110 READ #1,I;B$
120 IF A$#B$ THEN 210
130 IF END#1 THEN 190
140 FOR J=I TO 200
150 READ #1,J+1;A$,A1,A2,A3,A4,A5
160 PRINT #1,J;A$,A1,A2,A3,A4,A5
170 NEXT J
180 GOTO 60
190 PRINT #1,J;END
200 GOTO 60
210 NEXT I
220 GOTO 60
230 PRINT "DONE"
240 PRINT
250 END
```

The program segment loaded on SF key 6, shown above, allows you to *delete* specific employees from the file. Of course, SF keys 3 and 4 can be pressed anytime to give you an updated table of employees and their hours.

This data base program can be easily modified for your particular needs. The number of employees listed, for example, is limited only by the size of the file used to store the data. Inventory, payroll, or any other application that requires a large amount of data to be added, changed or deleted, can be incorporated in this sort of program.

Shown below are the tables generated by SF keys 3 and 4. The data stored on the file, MUSIC, respresents the number of hours worked by nine cashiers at a large musical instrument store.

```
EMPLOYEE         MON   TUES  WED   THURS  FRI

J.S. BACH         5     6     8     7     4
J. BRAHMS         8     8     8     8     8
L.V. BEETHOVEN    1     1     9     15    6
G. VERDI          6     5     6     5     6
R. STRAUSS        7     4     7     8     5
P. DUKAS          5     6     1     8     8
A. DVORAK         9     8     8     9     9
G. HOLST          10    2     5     4     3
I. STRAVINSKY     11    10    8     4     9


EMPLOYEE         HOURS

J.S. BACH         30
J. BRAHMS         40
L.V. BEETHOVEN    32
G. VERDI          28
R. STRAUSS        31
P. DUKAS          28
A. DVORAK         43
G. HOLST          24
I. STRAVINSKY     42

TOTALS           MON   TUES  WED   THURS  FRI   WEEK'S TOTAL

                  62    50    60    68     58     298
```

EXAMPLE 2-RAINFALL PROGRAM

The program shown below illustrates how data can be sorted and listed according to specific criteria. In particular, this program allows you to average the amount of rainfall that fell during a certain month in a particular area for up to 64 years. A String Variables ROM and a Matrix Operations ROM are required to run this program.

```
10 FIXED 2
20 DIM A[64],B[12],N[12],S[12],T[12],A$[6],B$[60],M$[144]
30 M$[1,54]="JANUARY  FEBRUARY MARCH    APRIL    MAY      JUNE"
40 M$[55,108]="JULY     AUGUST   SEPTEMBEROCTOBER  NOVEMBER DECEMBER"
50 FILES *,*,*,*,*,*,*,*,*,*,*
60 DISP "HOW MANY DATA FILES";
70 INPUT N
80 FOR I=1 TO N
90 DISP "NAME OF FILE";I;" 1-6 CHARS";
100 INPUT A$
110 B$[I*6-5,I*6]=A$
120 ASSIGN A$,I,X
130 IF X=0 THEN 170
140 PRINT A$;" NOT ASSIGNED."
150 DISP "GIVE NEW FILE";I;
160 GOTO 100
170 NEXT I
180 DISP "HOW MANY YEARS AGO";
190 INPUT Y
200 DISP "# OF MONTHS TO BE ANALYZED";
210 INPUT M
220 IF M>12 OR M<1 THEN 200
230 FOR I=1 TO M
240 B[I]=I
250 DISP I;". MONTH #";
260 INPUT B[I]
270 NEXT I
280 MAT N=ZER
290 MAT S=ZER
300 MAT T=ZER
310 FOR I=1 TO N
320 FOR J=1 TO M
330 IF END#I THEN 530
340 MAT READ # I,B[J];A
350 FOR Y1=65-Y TO 64
360 N[J]=N[J]+1
370 T[J]=T[J]+A[Y1]
380 S[J]=S[J]+A[Y1]↑2
390 NEXT Y1
400 NEXT J
410 NEXT I
420 FOR J=1 TO M
430 PRINT M$[B[J]*9-8,B[J]*9];" MEAN ";
440 IF T[J]=0 THEN 490
450 PRINT T[J]/(N[J]+(N[J]=0));
460 IF N[J]<2 THEN 500
470 PRINT " S.D. ";SQR((N[J]*S[J]-T[J])/(N[J]*(N[J]-1)))
480 GOTO 500
490 PRINT 0
500 PRINT
510 NEXT J
520 GOTO 540
530 DISP "DATA CANNOT SUPPORT THIS PROGRAM";
540 END
```

The program is run when one or more data files, of 12 records apiece (one for each month of each year), contain data. Since the 12 records can each hold 64 data elements† each file can contain 64 years of data. The number of years of data required can be changed as needed. For example, if only 10 years of data are stored on your data files, change the variable A, in line 20, from 64 characters to 10. Line 350 must be changed, in that case, as follows:

```
350 FOR Y1=11-Y TO 10
```

If you attempt to read more data than is physically present in the data file you are accessing, line 330 sets up a condition to branch to line 530 when the MAT READ# statement (line 340) encounters an end of record marker.

The program asks you to input the number of files you wish to access and, one by one, their names. Once you specify the number of years over which you want to take averages, the program counts back from the last data entry. In this way, averages can be taken over a short or long period of time. The number of months you are interested in must be input. After this, execute 1 for January, 2 for February, etc. Notice that any combination of months, up to 12, can be accessed to obtain the mean and standard deviation, month by month.

A sample printout for 12 months rainfall over a 35-year period in Lafayette, Colorado is shown below.

```
JANUARY   MEAN  1.01    S.D.  1.19
FEBRUARY  MEAN  0.70    S.D.  0.91
MARCH     MEAN  1.33    S.D.  1.60
APRIL     MEAN  2.05    S.D.  2.21
MAY       MEAN  1.55    S.D.  1.71
JUNE      MEAN  1.62    S.D.  1.85
JULY      MEAN  0.91    S.D.  0.97
AUGUST    MEAN  1.19    S.D.  1.29
SEPTEMBER MEAN  1.12    S.D.  1.39
OCTOBER   MEAN  1.57    S.D.  1.73
NOVEMBER  MEAN  1.37    S.D.  1.49
DECEMBER  MEAN  1.15    S.D.  1.37
```

† See Appendix B for a detailed discussion on how to estimate file size.

NOTES

APPENDIX A
MASS MEMORY STRUCTURE

The mass memory system is organized around user–defined areas of memory, called files, on the platter. Each platter can contain up to 768 files, depending on the size of each file. Files can be used to hold data (data files), programs (program files), or Special Function keys (key files).

As the user, you create these files, name them and — for data files — specify their size. Special versions of BASIC programming statements enable you to store (i.e., print) information on and retrieve (i.e., read) information from your mass memory files.

Each file contains one or more physical records. A record is the smallest addressable unit on the platter and contains 256 words of memory. Each record is actually a collection of one or more individual data items.

Although you must specify the size, in records, of a data file when you create it, the size of a program or key file is automatically determined; it is the number of records required to store the program. A file cannot be greater than 4752 records without exceeding the available storage space of the platter.

Do not confuse the terminology associated with the Model 30 calculator internal cassette system with that of the mass memory system. The calculator cassette system refers to a group of data items as a 'file'. Actually, in the mass memory terminology, it is a 'record', while the group of cassette files marked the same comprise what is equivalent to a mass memory 'file'.

DATA FILE STRUCTURE

As the user, you determine which method of data access best suits your needs; this decision is often based on the amount of available mass memory storage and the time required for your operations. An understanding of data file structure is therefore essential to make the best use of your system.

When working with data files, it is important to understand the difference between physical records and logical records. A physical record contains a fixed amount of storage space. (See Figure A-1.) In the mass memory system, a physical record is 256 16-bit words. (Eight bits equal one byte and two bytes equal one word.) A record is the smallest unit of data which can be accessed directly (i.e., randomly) by the system.

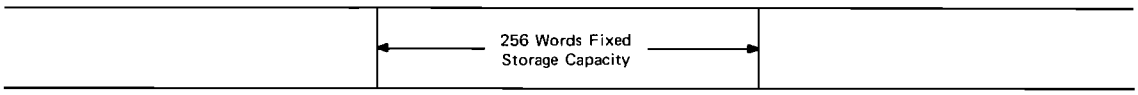


Figure A-1. A Physical Record

DATA FILE STRUCTURE

(Continued)

A logical record is a collection of items handled (i.e., stored or retrieved) at one time. (See Figure A-2.) This collection of items is specified by one or more READ# or PRINT# 'lists' of data elements, which define the logical records of the system. You must deal with this list, or logical record, as a unit.

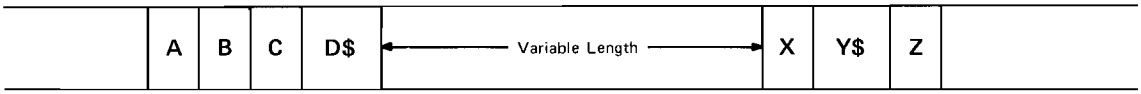


Figure A-2. A Logical Record

NOTE

Because the number of items which may appear on one BASIC line is limited, more than one READ# or PRINT# statement may be required to handle all the data of one logical group. In this case, the combined READ# or PRINT# lists define the logical record. (See footnote on page A-7.)

A logical record contains as many data items as necessary in a particular application to make up the working unit of data. For this reason, logical and physical records may or may not be related. A logical record can use a fraction of one physical record or a substantial number of physical records.

SERIAL FILE ACCESS

When working with data in the serial file access mode, you can define logical records to be any length. There is no correspondence between logical and physical records.† (See Figure A-3.)

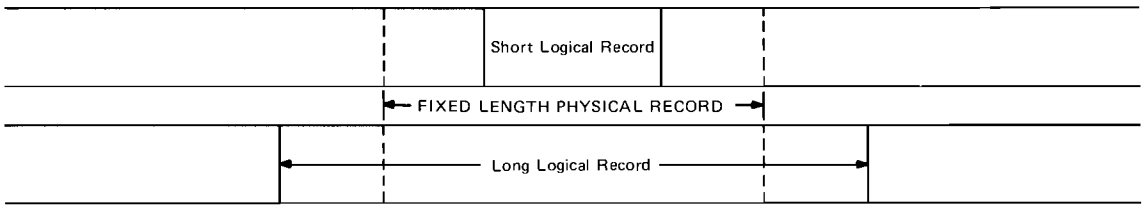


Figure A-3. A Logical Record Can Be Any Length in Serial File Access Mode

Logical records are stored next to each other, without any identifiable marker separations. All or part of the information stored originally can be retrieved in one READ# statement. The list of data elements read comprise a logical record, but this list does not have to be identical to the list originally printed on the file. In other words, logical records are defined as you print or read them. In order to retrieve all of the information stored, however, your READ# statement data list must match†† the PRINT# statement data lists stored previously.

The beginning of a physical record is the only point where direct access is possible. Storage space is utilized with maximum efficiency when serial PRINT# statements are used, since logical records are packed solidly and no space is left unused between them.

† The only point of 1:1 correspondence between physical and logical records is at the beginning of the file, where the beginning of the first logical record coincides with the beginning of the first physical record.

†† These data lists must have identical elements as far as size, type and order are concerned. The names you assign to these elements can still vary.

RANDOM FILE ACCESS

When working with data in the random file access mode, you must specify which record within a file you want to access. There is a 1:1 correspondence between logical and physical records. In this case, while any logical record can be (and generally is) shorter than a physical record, a logical record can *never* be longer than a physical record (256 words).

Since the beginning of a logical record coincides with the beginning of a physical record, short logical records do not utilize storage space effectively. Only the space required for storage is used; the rest of the space in the physical record is not used.

Of course, the advantage of using random file access is that every logical record is directly accessible, in any order.

SERIAL VS. RANDOM FILE ACCESS

As mentioned before, you decide on which method of data accessing to use for your particular needs. This decision is usually not made easily, because of the inherent advantages and disadvantages of both methods. More efficient storage space utilization must be sacrificed for a shorter access time, and vice versa. *Once your decision has been made, it is difficult to change later; so make your decision carefully.*

The following table summarizes the advantages and disadvantages of accessing data from a file serially and randomly.

Table A-1. Serial vs. Random File Access

Feature	Serial File Access	Random File Access
Storage Efficiency	GOOD-Data is packed solidly	VARIES-Poorer for short logical records
Access Time	VARIES-Longer for higher-numbered records	GOOD-Directly to any record
Logical Record Length	Any length that fits into a file	Less than or equal to physical record length

END OF RECORD (EOR) MARKERS

There are two types of end of record (EOR) markers: logical end of record (LEOR) and physical end of record (PEOR). These coincide with logical and physical records respectively. The LEOR marker indicates the end of data in a physical record, while the PEOR marker indicates the physical end of the record itself. An LEOR marker is actually stored in one word of each mass memory record, but PEOR markers are system-generated; they do not take up space on records. For purposes of discussion, they are called 'markers' although, technically, they are system–detected conditions.

DATA FILE STRUCTURE

(Continued)

LEOR markers are placed in certain positions and stored in the system in two ways. After execution of every PRINT# statement, an LEOR marker is placed automatically after the last item in the PRINT# data item list. If a serial PRINT# statement instructs the system to print data over an existing PEOR marker, the system generates an LEOR marker internally near the end of the physical record in which it encountered the PEOR marker.

Figure A-4 is a representation of a three–record file, MOM. When the file is opened and a FILES statement is executed, the pointer is positioned at the beginning of the first physical record.

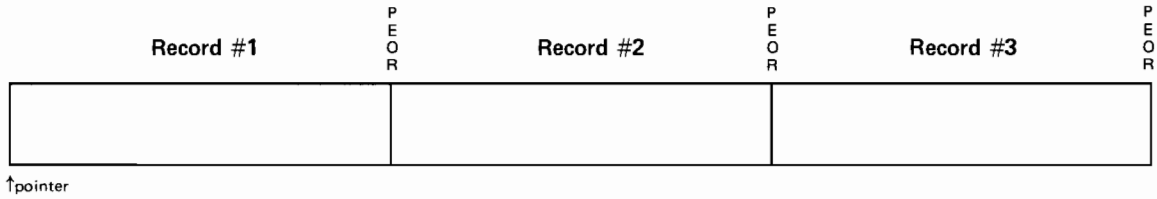


Figure A-4. "MOM"

```
10 OPEN "MOM",3
20 FILES MOM
```

When a serial PRINT# statement is executed, an LEOR marker is placed immediately after the last data item.

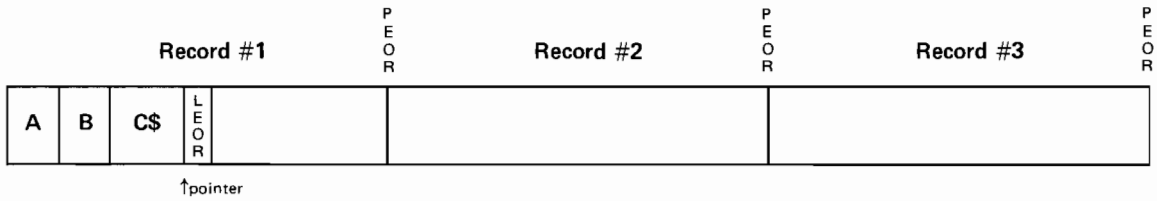


Figure A-5. LEOR Marker Placed after Last Data Item

```
30 PRINT #1;A,B,C$
```

Executing another serial PRINT# statement stores the new data list from the pointer position, moving the LEOR marker to the end of the last current data item.

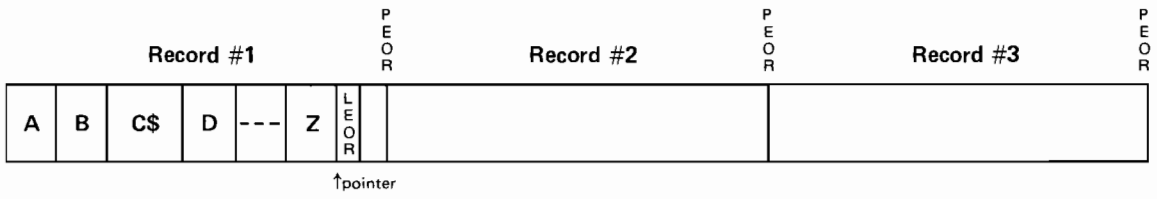


Figure A-6. LEOR Marker Is Moved

```
40 PRINT #1;D,...,Z
```

If the remaining space in a physical record is inadequate for a data element in a subsequent PRINT# list, an LEOR marker is placed internally as close to the PEOR marker in record 1 as possible (i.e., after the last data element that fits in the record). The remainder of the data list is stored on the next physical record.

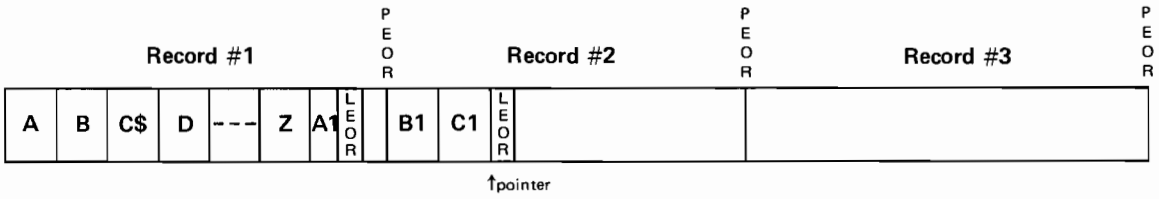


Figure A-7. Insufficient Space on One Physical Record

```
50 PRINT #1;A1,B1,C1
```

Use a READ# statement to reposition the pointer to the beginning of the first physical record. (This can also be accomplished by executing a FILES statement.)

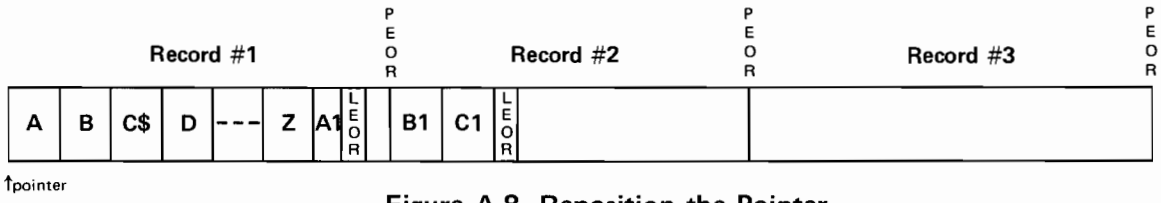


Figure A-8. Reposition the Pointer

```
60 READ #1,1
```

Then use another serial READ# statement to retrieve the data in the first logical record.

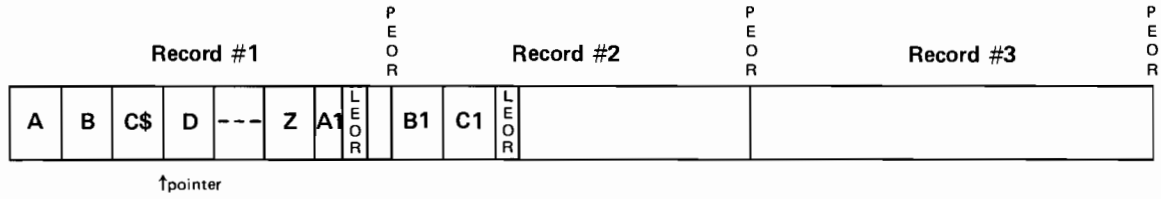


Figure A-9. Read Data in First Logical Record

```
70 READ #1;A,B,C$
```

DATA FILE STRUCTURE

(Continued)

Another serial READ# statement retrieves data from the position of the pointer.

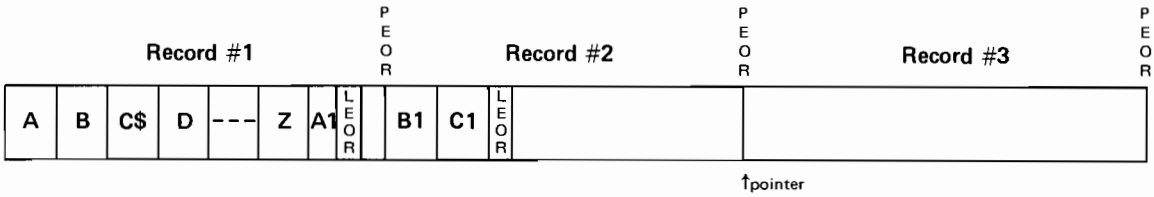


Figure A-10. Read Data in Rest of the File

```
80 READ #1;D,...,Z,A1,B1,C1,X1
```

Notice that when the LEOR marker in the first physical record is encountered, the system automatically skips over the marker and continues to read the data stored in the second physical record. If the READ# data item list includes more variables than there are data items stored, when the second LEOR marker is encountered, the system attempts to read the data (in this case, X1) in the next physical record (record 3). There is no data in the third physical record, so an error message is returned.

All of the data in this file can be retrieved as one logical record, even though it was stored as three logical records. After repositioning the pointer to the beginning of the file, execute a serial READ# statement that includes all of the data items in the list. That is,

```
90 READ #1;D,...,Z,A1,B1,C1
```

An LEOR marker is also placed as a result of a random PRINT# statement. Figure A-11 is a representation of another three–record file, DAD. Once again, when the FILES statement is executed, the pointer is positioned at the beginning of the first record in the file.

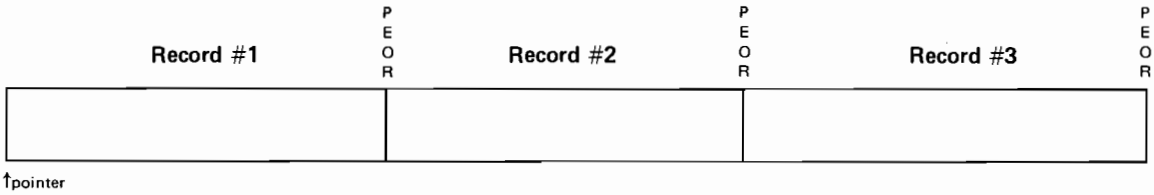


Figure A-11. "DAD"

```
10 OPEN "DAD",3
20 FILES DAD
```

As in the serial mode, when a random PRINT# statement is executed, the LEOR marker is placed immediately after the last data item.

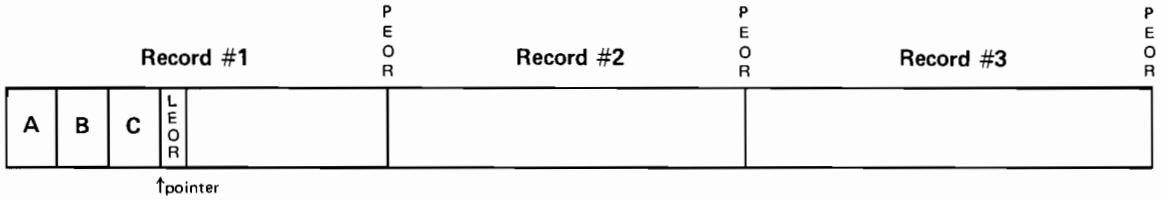


Figure A-12. LEOR Marker Placed after Last Data Item

```
30 PRINT #1,1;A,B,C
```

Executing another random PRINT# statement stores the new data list from the beginning of the physical record specified.

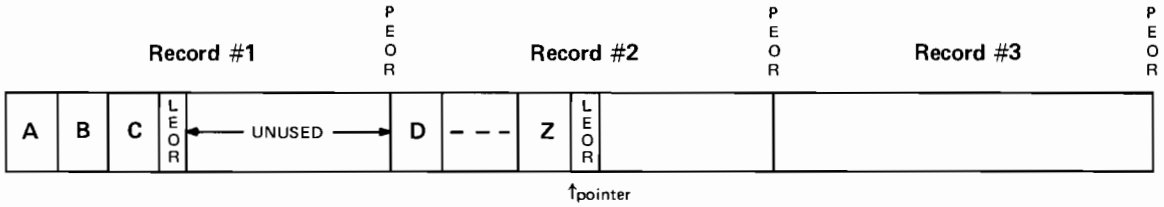


Figure A-13. Printing Another Logical Record

```
40 PRINT #1,2;D,...,Z
```

The space between the end of the first physical record and the LEOR marker in that record is unused.†

Of course, this second data list could have been stored in the third (or any) record, instead of the second record, as shown. Random PRINT# statements are used to store data in specified records and can be executed in any order. An LEOR marker is placed after the last item in each data list unless the data exactly fills the physical record. In this case, although an LEOR marker is not stored, the system acts as if it were.

† In general, this space can be used by executing a combination of serial and random PRINT# statements. Use the first random PRINT# statement to store data from the beginning of a specific record. A serial PRINT# statement stores data items over the LEOR marker placed after the random PRINT# data item list.

For example, consider a logical record that contains six full precision items, each of which is a 12–digit number containing ten decimal places. Since only four of these numbers fit completely into a BASIC line, two PRINT# statements must be executed. In this example, the first PRINT# stores four items in the fifth record of a file and the second PRINT# stores the last two items immediately after the first four.

```
PRINT #1,5;43.5476809782,32.3523640987,20.1423549974,19.2647516208
PRINT #1;55.3254709806,59.2645773841
```

These data items can be retrieved in six variables by executing one random READ# statement.

```
READ #1,5;A,B,C,D,E,F
```

Multiple READ# statements can be used in the same manner for longer logical records.

DATA FILE STRUCTURE

(Continued)

To read the data in any given record, execute a random READ# statement, specifying the particular record.

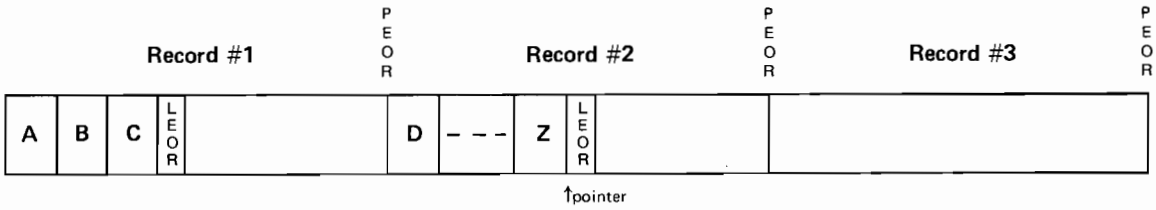


Figure A-14. Data in the Second Record Is Read

```
50 READ #1,2;D,...,Z
```

Notice that the pointer does not have to be repositioned before the random READ# statement is executed; the pointer is moved to the beginning of the specified record before reading occurs. Similarly, to read the data in the first record, execute a random READ# statement, specifying record number 1. The pointer moves to the beginning of the first record before the data is read.

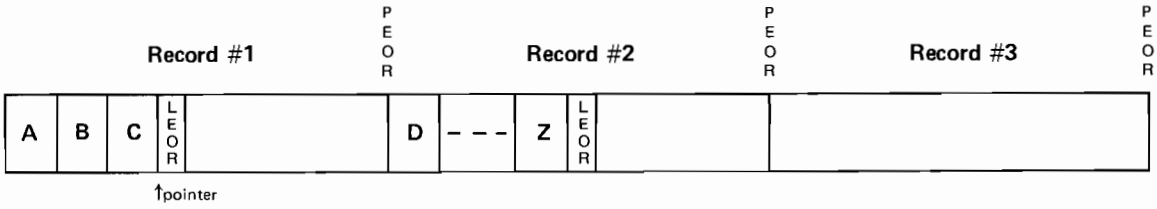


Figure A-15. Data in the First Record Is Read

```
60 READ #1,1;A,B,C
```

Once data has been stored, randomly or serially, it can be retrieved by either a random or a serial READ# statement. Remember that when an LEOR or PEOR marker is encountered by a serial READ# statement, the system attempts to read the next data item in the following physical record. When an LEOR or PEOR marker is encountered by a *random* READ# statement, however, an end of record condition is detected. This results in an error message (or in program branching to another line of the program when an IF END# statement is previously executed – see "IF END# Statement", page 3-24).

END OF FILE (EOF) MARKERS

There are two types of end of file (EOF) markers: logical end of file (LEOF) and physical end of file (PEOF). The LEOF marker is actually stored on the mass memory files, while the PEOF marker is system–generated and indicates the end of space allocated to the file.

During execution of an OPEN command, LEOF markers are placed in the first word of every physical record of the file. These markers disappear, however, as soon as data is stored on the records. LEOF markers are also placed when a PRINT# statement, which includes the optional parameter, END, is executed.

When END is part of the PRINT# statement, the LEOR marker previously and automatically placed at the end of the specified logical record is replaced by an LEOF marker.

In Figure A-16, two of a file's three records contain data and the third record is empty. An LEOF marker is placed in the first word of each record when the file is opened. After data is stored in the first two records, however, the LEOF markers in these records are either moved or disappear, while the third record LEOF marker remains.

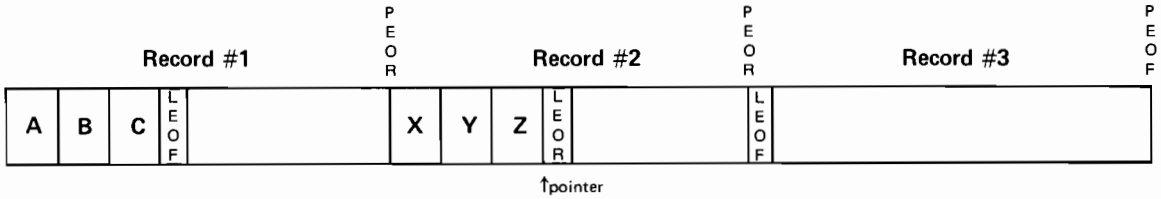


Figure A-16. Data Is Printed in the First and Second Records

```
10 OPEN "SONNY",3
20 FILES SONNY
30 PRINT #1,1;A,B,C,END
40 PRINT #1,2;X,Y,Z
```

Notice that by executing line 30 (above), an LEOF marker is placed after the last data item in record #1. As mentioned previously, the pointer is positioned at the LEOR marker in the second record after the data in the second record is read. (See Figure A-17.)

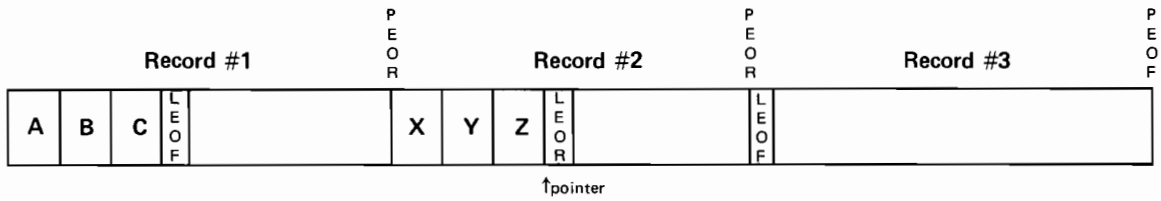


Figure A-17. Data in the Second Record Is Read

```
50 FILES SONNY
60 READ #1,2;X,Y,Z
```

If a serial READ# statement is executed at this point, the system encounters the LEOR marker and automatically moves to the beginning of the next physical record to attempt to read another data item. Here, encountering an LEOF marker establishes an end of file condition. Once again, this results in an error message (or causes the program to branch to another line of the program when an IF END# statement is previously executed – see "IF END# Statement", page 3-16).

Similarly, if a random READ# statement is executed from the beginning of the third record, the system encounters the LEOF marker, establishing an end of file condition (see "IF END# Statement", page 3-24).

The end of file condition is detected most often when executing a READ# statement, but it is also detected when an PRINT# statement attempts to store data beyond the actual physical end of space allocated to the file.

DATA FILE STRUCTURE

(Continued)

A summary of end of file and end of record markers is shown in the table below.

Table A-2. EOR and EOF Markers

Marker	Where Placed	When and How Established
PEOR	At the end of each physical record in a file	Automatically, when a file is opened
LEOR	At the end of every data item list	Automatically, after each PRINT# statement, unless the optional END parameter is included
PEOF	At the end of each physical file	Automatically, when a file is opened
LEOF	a. In the first word of every physical record	a. Automatically, when a file is opened
	b. At the end of a data item list under certain conditions (see next column)	b. By executing a PRINT# statement including the optional END parameter

EOR AND EOF CONDITIONS

End of record and end of file conditions are established and can be detected according to the following detection table. Note that EOR and EOF conditions are never established when any PRINT# statement encounters logical EOR or logical EOF markers; these markers are simply printed over.

Table A-3. EOR and EOF Conditions

	LEOR	LEOF	PEOR	PEOF
Serial PRINT#				★
Serial READ#		★		★
Random PRINT#			★	★
Random READ#	★	★	★	★

PLATTER STRUCTURE

The platter used in the mass memory system is an aluminum alloy disc, slightly larger than a standard long–playing phonograph record (14" diameter). Bonded onto both sides of the platter is a ferromagnetic iron oxide which has magnetic characteristics similar to magnetic tape. Two wide–temperature–range read–write heads in the mass memory drive are used to store and retrieve information on either side of the platter (see Figure A-19).

Data is stored on a platter in concentric tracks. Each platter has 406 tracks (203 on each side of the platter). These tracks are numbered from 0 to 405. The upper surface of the platter contains tracks 0 through 202; the lower surface contains tracks 203 through 405. Tracks 0–7 and 404–405 comprise the 'system area' and are reserved for system use; the remaining tracks comprise the 'user area' of the platter.

The platter is also subdivided into 12 pie–shaped physical records on each surface of the platter, numbered from 0 to 11. A physical record is 256 words. A CATALOG command specifies the exact location of a given record by listing its track number and record number under the headings, TRACK and RECORD.

12 records X (406 total tracks – 10 system tracks) = 4752 available records/platter

4752 records/platter X 256 words/record ≅ 1.2 million available words/platter

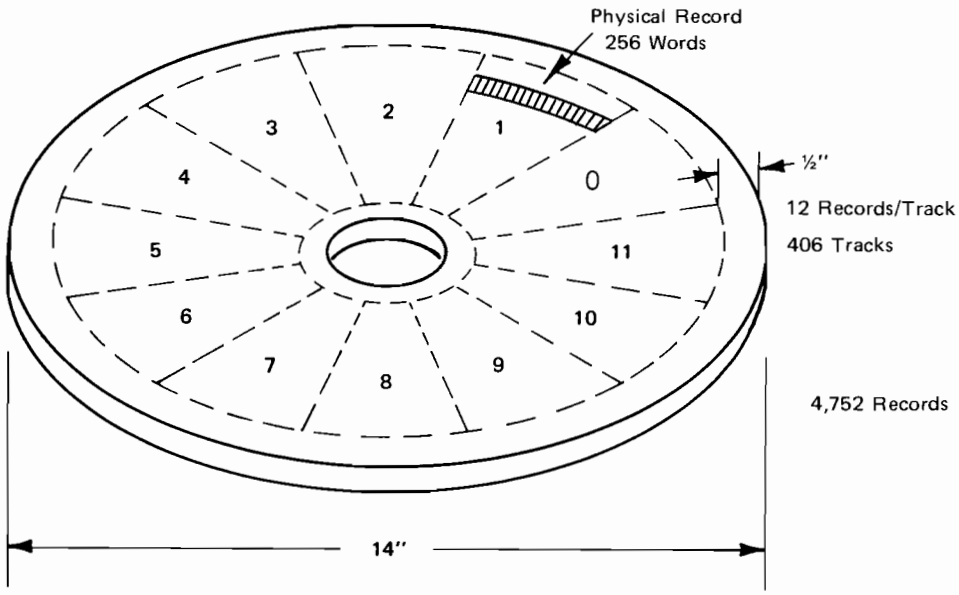


Figure A-18. Mass Memory System Platter

PLATTER STRUCTURE

(Continued)

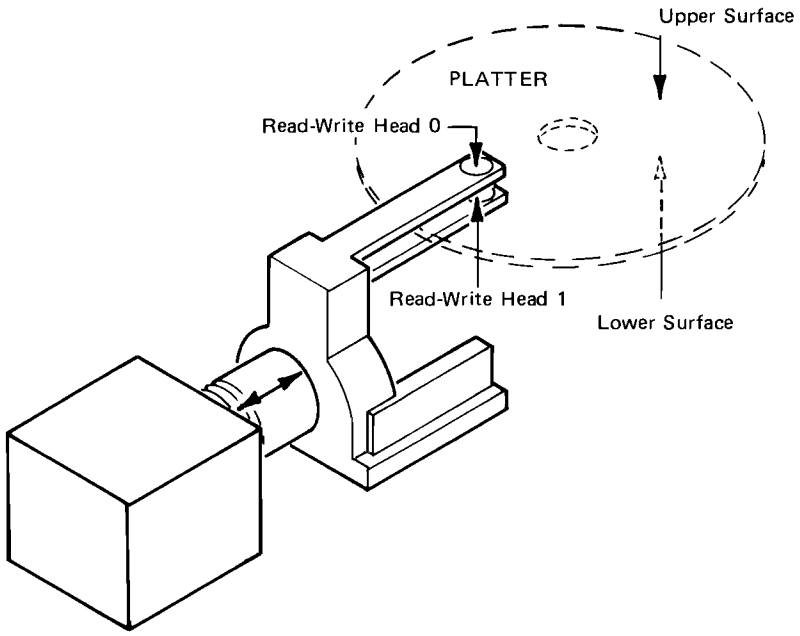


Figure A-19. Storing and Retrieving Data

APPENDIX B
STORAGE REQUIREMENTS

The efficacity of your mass memory system can be improved if you accurately estimate the size of the files needed to store information. To this end, this appendix will help you estimate the space required to store programs and data.

PROGRAM SIZE

As mentioned in the "SAVE Command" discussion (page 2-3), the mass memory system opens a program file large enough to accommodate the program in the calculator memory. If the number of words the program uses is not a multiple of 256 (it rarely is), the mass memory system rounds the number of physical records reserved in the file to the next whole number. While a program of 255 words requires one physical record, a program of 257 words requires two physical records.

DATA STORAGE

The following table lists the number of words of memory required to store full, split and integer precision data elements and string variables. Strings and numbers can be mixed within a physical record, as long as each item fits within the bounds of the record.

Table B-1. Data Storage Space

Type of Data	# of Words/Data Item	Max. # of Data Items/ Physical Record (256 Words)
full precision	4	64
split precision	2	128
integer precision	2†	128
string variable	see below	see below

NOTE

The type of data retrieved, determined by READ# and MAT READ# statements, can differ from the type of data stored, determined by PRINT# and MAT PRINT# statements. Data stored with full precision accuracy, for example, can be retrieved will full, split or integer precision. An error message is displayed if you attempt to convert a full or split precision number greater than |32,767| to an integer precision number.

† The mass memory system reserves two words of memory for integer precision accuracy, rather than one word, as reserved in the Model 30 calculator system.

DATA STORAGE

(Continued)

The space required to store a string variable is calculated in this manner:

- Divide the number of characters in the string by 2.
- Round the result to the next whole number.
- Add 1 to that result.

The maximum number of strings that can be stored in a physical record is 128 1– or 2–character strings. Conversely, since a string can contain up to 255 characters, a physical record can contain one 255–character (maximum length) string (129 words) plus one 252–character string (127 words). Eight 62–character strings also fill a physical record completely.

APPENDIX C
INCREASING AVAILABLE MEMORY

DAVTP COMMAND

The system area of each platter contains an 'availability table'. This table keeps a record of all the unused space on the platter. Whenever a file has to be created, by either an OPEN or a SAVE command, the system searches the availability table to find a space large enough for this new file. After the file is created, the space it requires is removed from the table. Similarly, when a file is killed, the space used by the file is returned to this table. As a result of creating and killing files over a period of time, the availability table contains a list of available spaces. This list is not necessarily in size or location order, nor are adjacent blocks of unused space combined to make larger spaces. For this reason, ERROR 95 (available storage space exceeded) may result when a SAVE or OPEN command is attempted although the catalog listing indicates that there is enough total storage space left for the new file.

The DAVTP command (data availability table pack) restructures the availability table by location and combines any adjacent spaces into larger spaces.

Syntax:

DAVTP

DAVTP takes up to three minutes to execute. Calculator memory, however, is erased when this command is executed. Executing DAVTP resets the system to unit 0. This command can be used as often as desired to reorganize the availability table.

At some point the available space may be fragmented in so many small areas all over the platter that no further files of any reasonable size can be created; the DAVTP has no effect when this happens. The REPACK procedure (see next page) can be used to combine the remaining spaces.

REPACK PROCEDURE

The files on a platter can be moved together by performing the REPACK procedure, described below. The unused space on the platter is consolidated into one large space, which can then be used for storing additional files.

Insert the mass memory system tape cassette in the calculator and execute:

LOAD BIN 80

After the display returns, key in and execute: UNIT N (where N is the unit number of the platter you wish to repack). This step can be ignored if your platter is designated as unit 0.

Key in the word, REPACK, and execute it.

The REPACK procedure is performed in 10 minutes or less. Do not rewind the tape or remove the tape cassette from the calculator until control returns, because the REPACK procedure uses three cassette files which are automatically loaded and executed during the course of the procedure.

If a cassette or transport error (ERROR 58 or 59) occurs while the REPACK procedure is being performed, the information contained on your platter remains intact. Although the availability table and catalog listings may not reflect current file *locations* at this point, data and program file contents are still unaltered. Check positioning of the tape cassette and make sure the tape transport door is closed securely. Then simply perform the procedure from the beginning. If this error occurs again, perform the procedure once more from the beginning, substituting LOAD BIN 83 for the first step.

APPENDIX D
ACCESS TIME AND BOOTSTRAPS

The time needed to transfer programs or data between your calculator and the mass memory drive varies with the length of the program or with the amount of data. The following graphs (Figures D–1 through D–3) show approximate times required for transfer of information.

Figure D–1 shows the time required to transfer a program from a platter to a calculator in your mass memory system. Programs are transferred at a rate of about 1,000 16–bit words per second. The transfer of a program from a calculator to a platter, using the SAVE command, requires 10% to 25% more time.

Figure D–2 shows the typical time required to transfer one full precision number. This time is based on the number of data items transferred with one command. The transfer time per item decreases as the number of data items in one PRINT# or READ# statement increases. It takes less time to execute:

```
100 PRINT #1;A,B,C,D,E
```

than it does to execute:

```
100 FOR I=1 TO 5
110 PRINT #1;A[I]
120 NEXT I
```

While the time needed to execute the first example (above) is 75 milliseconds, the second example takes 250 milliseconds. This is because overhead time associated with each PRINT# statement is constant. If this time is spread over more data items, the transfer time per item is proportionally less.

Figure D–3 shows the time needed to transfer matrices. The transfer time per item is very long when the array size is less than 5. In those cases, it is faster to use a normal PRINT# or READ# statement. For example, MAT PRINT# 1; A, where A is a one element array, takes 180 milliseconds to execute. Notice that these curves are essentially horizontal for 40 or more elements per statement. Therefore, there is almost no time to be gained by transferring more than 40 elements per statement.

The statements, command and function stored in the mass memory are:

PRINT#	UNIT
READ#	TYP

Other statements and commands are stored in the bootstraps on the platter and must be transferred into the calculator each time they are executed. For this reason, they take more time to execute than the above commands.

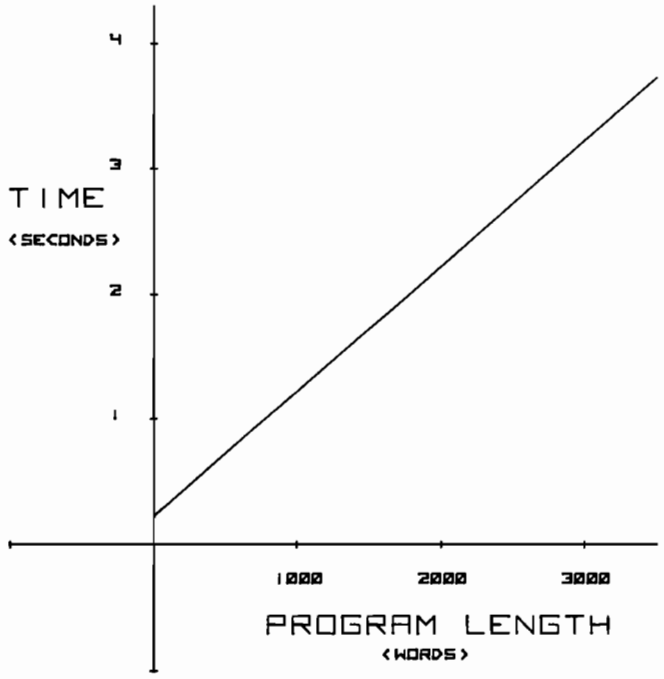


Figure D–1. Transfer of Programs (GET or CHAIN Command)

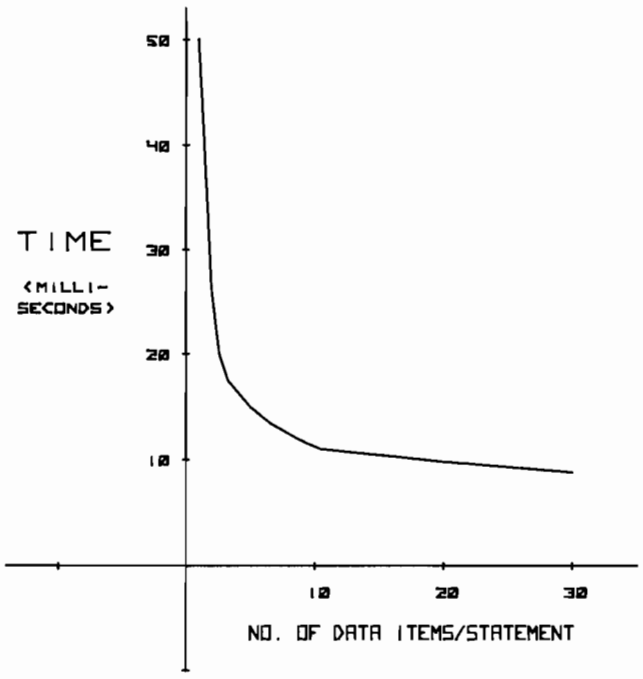


Figure D–2. Data Element Transfer Time

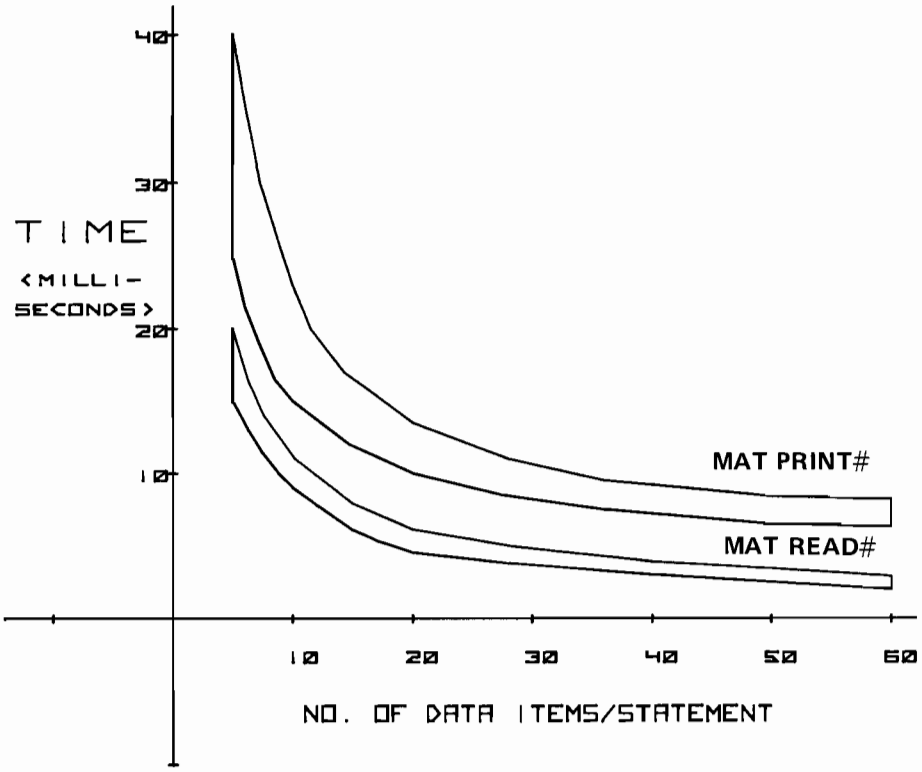


Figure D–3. Matrix Element Transfer Time

APPENDIX E
SUMMARY OF MASS MEMORY SYNTAXES

ASSIGN "file name†", file number, return variable [,"protection code†"]

Assigns a file name to a position in the previous FILES statement.

CAT

Lists information about every file on the platter.

CHAIN "file name†" [1st line number [,2nd line number]]

Loads a program from the platter to the calculator, retaining current values of variables.

DAVTP

Restructures availability table.

DBYTE variable, string name

Converts value of specified variable to its binary equivalent character.

DCOPY "1st file name" [,unit number] TO "2nd file name" [,unit number]

Duplicates contents of one data file into another.

DEXP variable, string name

Converts value of specified variable into a 4–digit character string with leading zeros.

DFDUMP "file name"

Stores specified data file presently on the platter to the calculator internal cassette(s).

DFLOAD "file name"

Loads data presently on calculator internal cassette(s) to a specific file on the platter.

DGET "file name" [0]

Loads source (non–compiled) program into the calculator and checks for syntax errors.

DREN "old file name" TO "new file name" [,"protection code"]

Changes the name of any file.

FILES file name or * [,file name or *] [,...]

Declares which files are to be used.

GET "file name†" [1st line number [,2nd line number]]

Loads a program from the platter to the calculator.

GET KEY "file name†"

Loads Special Function key definitions from a specified file of the platter to the calculator Special Function keys.

IF END# file number THEN line number

Sets up an exit procedure which branches the program to a specific line number when an end of file or end of record condition is encountered.

KILL "file name† " [,"protection code†"]

Erases the named file from the platter.

MAT PRINT# file number [,record number] ; list of matrix variables

Prints an entire matrix onto a specified record or file.

MAT READ# file number [,record number] ; list of matrix variables

Reads a matrix from a specified record or file.

OPEN "file name†", number of records

Creates a data file with a specified number of physical records and assigns it a name.

(Random) PRINT# file number, record number; list [,END]
PRINT# file number, record number [;END]

Prints data on a file from the beginning of a specified physical record.

(Serial) PRINT# file number; list [,END]
PRINT# file number; END

Prints data on a file after the last item previously read or printed or at the beginning of the file.

PRO "file name", "protection code"

Assigns a protection code to a specific file.

(Random) READ# file number, record number [;list]

Reads data from a specified record in a file.

(Serial) READ# file number; list

Reads data from a specified file, starting after the last item printed or read.

SAVE "file name†" [,1st line number [,2nd line number]]

Stores an entire program or parts of it onto a specified file of the platter.

SAVE KEY "file name†"

Prints Special Function key definitions onto a specified file of the platter.

TYP file number or TYP (–file number)

Identifies the type of the next item in a specified file.

UNIT unit number

Specifies the platter to be used for the subsequent commands.

† This parameter can be a string variable. When a string variable is used, the quotation marks (") surrounding it must be removed.

NOTES

INDEX

A

B

C

D

E

Q

R

S

T

U

ELECTRONIC

SALES & SERVICE OFFICES

UNITED STATES

ALABAMA
8290 Whitesburg Dr., S.E.
P.O. Box 4207
Huntsville 35802.
Tel: (205) 881-4591
TWX: 810-726-2204

ARIZONA
2336 E. Magnolia St.
Phoenix 85034
Tel: (602) 244-1361
TWX: 910-951-1330

2424 East Aragon Rd.
Tucson 85706
Tel: (602) 889-4661

CALIFORNIA
1430 East Orangethorpe Ave.
Fullerton 92631
Tel: (714) 870-1000
TWX: 910-592-1288

3939 Lankershim Boulevard
North Hollywood 91604
Tel: (213) 877-1282
TWX: 910-499-2170

6515 Arizona Place
Los Angeles 90045
Tel: (213) 776-7500
TWX: 910-328-6148

1101 Embarcadero Road
Palo Alto 94303
Tel: (415) 327-6500
TWX: 910-373-1280

2220 Watt Ave.
Sacramento 95825
Tel: (916) 482-1463
TWX: 910-367-2092

9606 Aero Drive
P.O. Box 23333
San Diego 92123
Tel: (714) 279-3200
TWX: 910-335-2000

COLORADO
5600 South Ulster Parkway
Englewood 80110
Tel: (303) 771-3455
TWX: 910-935-0705

CONNECTICUT
12 Lunar Drive
New Haven 06525
Tel: (203) 389-6551
TWX: 710-465-2029

FLORIDA
P.O. Box 24210
2806 W. Oakland Park Blvd.
Ft. Lauderdale 33307
Tel: (305) 731-2020
TWX: 510-955-4099

P.O. Box 13910
6177 Lake Ellenor Dr.
Orlando, 32809
Tel: (305) 859-2900
TWX: 810-850-0113

GEORGIA
P.O. Box 28234
450 Interstate North
Atlanta 30328
Tel: (404) 436-6181
TWX: 810-766-4890

HAWAII
2875 So. King Street
Honolulu 96814
Tel: (808) 955-4455

ILLINOIS
5500 Howard Street
Skokie 60076
Tel: (312) 677-0400
TWX: 910-223-3613

INDIANA
3839 Meadows Drive
Indianapolis 46205
Tel: (317) 546-4891
TWX: 810-341-3263

IOWA
1902 Broadway
Iowa City 52240
Tel: (319) 338-9466
Night: (319) 338-9467

LOUISIANA
P. O. Box 840
3239 Williams Boulevard
Kenner 70062
Tel: (504) 721-6201
TWX: 810-955-5524

MARYLAND
6707 Whitestone Road
Baltimore 21207
Tel: (301) 944-5400
TWX: 710-862-9157

20010 Century Blvd.
Germantown 20767
Tel: (31) 428-0700

P.O. Box 1648
2 Choke Cherry Road
Rockville 20850
Tel: (301) 948-6370
TWX: 710-828-9684

MASSACHUSETTS
32 Hartwell Ave.
Lexington 02173
Tel: (617) 861-8960
TWX: 710-326-6904

MICHIGAN
23855 Research Drive
Farmington 48024
Tel: (313) 476-6400
TWX: 810-242-2900

MINNESOTA
2459 University Avenue
St. Paul 55114
Tel: (612) 645-9461
TWX: 910-563-3734

MISSOURI
11131 Colorado Ave.
Kansas City 64137
Tel: (816) 763-8000
TWX: 910-771-2087

148 Weldon Parkway
Maryland Heights 63043
Tel: (314) 567-1455
TWX: 910-764-0830

***NEVADA**
Las Vegas
Tel: (702) 382-5777

NEW JERSEY
W. 120 Century Rd.
Paramus 07652
Tel: (201) 265-5000
TWX: 710-990-4951

NEW MEXICO
P.O. Box 8366
Station C
6501 Lomas Boulevard N.E.
Albuquerque 87108
Tel: (505) 265-3713
TWX: 910-989-1665

156 Wyatt Drive
Las Cruces 88001
Tel: (505) 526-2485
TWX: 910-983-0550

NEW YORK
6 Automation Lane
Computer Park
Albany 12205
Tel: (518) 458-1550
TWX: 710-441-8270

1219 Campville Road
Endicott 13760
Tel: (607) 754-0050
TWX: 510-252-0890

New York City
Manhattan, Bronx
Contact Paramus, NJ Office
Tel: (201) 265-5000
Brooklyn, Queens, Richmond
Contact Woodbury, NY Office
Tel: (516) 921-0300

82 Washington Street
Poughkeepsie 12601
Tel: (914) 454-7330
TWX: 510-248-0012

39 Seginaw Drive
Rochester 14623
Tel: (716) 473-9500
TWX: 510-253-5981

5858 East Molloy Road
Syracuse 13211
Tel: (315) 454-2486
TWX: 710-541-0482

1 Crossways Park West
Woodbury 11797
Tel: (516) 921-0300
TWX: 510-221-2168

NORTH CAROLINA
P.O. Box 5188
1923 North Main Street
High Point 27262
Tel: (919) 885-8101
TWX: 510-926-1516

SOUTH CAROLINA
6941-0 N. Trenholm Road
Columbia 29260
Tel: (803) 782-6493

OHIO
16500 Sprague Road
Cleveland 44130
Tel: (216) 243-7300
Night: 243-7305
TWX: 810-423-9431

330 Progress Rd.
Dayton 45449
Tel: (513) 859-6202
TWX: 810-459-1925

6665 Busch Blvd.
Columbus 43229
Tel: (614) 846-1300

OKLAHOMA
P.O. Box 32008
Oklahoma City 73132
Tel: (405) 721-0200
TWX: 910-830-6862

OREGON
17890 SW Boones Ferry Road
Tualatin 97062
Tel: (503) 620-3350
TWX: 910-467-8714

PENNSYLVANIA
111 Zeta Drive
Pittsburgh 15238
Tel: (412) 782-0400
Night: 782-0401
TWX: 710-795-3124

1021 8th Avenue
King of Prussia Industrial Park
King of Prussia 19406
Tel: (215) 265-7000
TWX: 510-660-2670

RHODE ISLAND
873 Waterman Ave.
East Providence 02914
Tel: (401) 434-5535
TWX: 710-381-7573

***TENNESSEE**
Memphis
Tel: (901) 274-7472

TEXAS
P.O. Box 1270
201 E. Arapaho Rd.
Richardson 75080
Tel: (214) 231-6101
TWX: 910-867-4723

P.O. Box 27409
6300 **Westpark Drive**
Suite 100
Houston 77027
Tel: (713) 781-6000
TWX: 910-881-2645

231 **Billy Mitchell Road**
San Antonio 78226
Tel: (512) 434-4171
TWX: 910-871-1170

UTAH
2890 South Main Street
Salt Lake City 84115
Tel: (801) 487-0715
TWX: 910-925-5681

VIRGINIA
P.O. Box 9854
2914 Hungary Springs Road
Richmond 23228
Tel: (804) 285-3431
TWX: 710-956-0157

***WEST VIRGINIA**
Charleston
Tel: (304) 345-1640

WISCONSIN
9431 W. Beloit Road
Suite 117
Milwaukee 53227
Tel: (414) 541-0550

FOR U.S. AREAS NOT LISTED:
Contact the regional office nearest you: Atlanta, Georgia . . . North Hollywood, California . . . Paramus, New Jersey . . . Skokie, Illinois. Their complete addresses are listed above.
*Service Only

CANADA

ALBERTA
Hewlett-Packard (Canada) Ltd.
11748 Kingsway Ave.
Edmonton T5G 0X5
Tel: (403) 452-3670
TWX: 610-831-2431

Hewlett-Packard (Canada) Ltd.
915-42 Avenue S.E.
Calgary T2G 1Z1
Tel: (403) 262-4279

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

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

NOVA SCOTIA
Hewlett-Packard (Canada) Ltd.
2745 Dutch Village Rd.
Halifax B3L 4G7
Tel: (902) 455-0511
TWX: 610-271-4482

ONTARIO
Hewlett-Packard (Canada) Ltd.
1785 Woodward Dr.
Ottawa K2C 0P9
Tel: (613) 225-6530
TWX: 610-562-8968

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
Mississauga L4V 1L9
Tel: (416) 678-9430
TWX: 610-492-4246

QUEBEC
Hewlett-Packard (Canada) Ltd.
275 Hymus Boulevard
Pointe Claire H9R 1G7
Tel: (518) 561-6520
TWX: 610-422-3022
Telex: 05-821521 HPCL

Hewlett-Packard (Canada) Ltd.
2376 Galvani Street
Ste-Foy G1N 4G4
Tel: (418) 688-8710

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

CENTRAL AND SOUTH AMERICA

ARGENTINA
Hewlett-Packard Argentina S.A.C.e.I
Lavalle 1171 - 3°
Buenos Aires
Tel: 35-0436, 35-0627, 35-0341
Telex: 012-1009
Cable: HEWPACK ARG

BOLIVIA
Stambuk & Mark (Bolivia) LTDA.
Av. Mariscal, Santa Cruz 1342
La Paz
Tel: 40626, 53163, 52421
Telex: 3560014
Cable: BUKMAR

BRAZIL
Hewlett-Packard Do Brasil
I.E.C. Ltda.
Rua Frei Caneca 1119
01307-**Sao Paulo**-SP
Tel: 288-7111, 287-5858
Telex: 309151/2/3
Cable: HEWPACK Sao Paulo

Hewlett-Packard Do Brasil
I.E.C. Ltda.
Praca Dom Feliciano, 78
90000-**Porto Alegre**-RS
Rio Grande do Sul (RS) Brasil
Tel: 25-8470
Cable: HEWPACK Porto Alegre

Hewlett-Packard Do Brasil
I.E.C. Ltda.
Rua da Matriz, 29
20000-**Rio de Janeiro**-GB
Tel: 266-2643
Telex: 210079 HEWPACK
Cable: HEWPACK Rio de Janeiro

CHILE
Héctor Calcagni y Cia, Ltda.
Casilla 16.475
Santiago
Tel: 423 96
Cable: CALCAGNI Santiago

COLOMBIA
Instrumentación
Henrik A. Langebaek & Kier S.A.
Carrera 7 **No. 48-59**
Apartado Aéreo 6287
Bogota, 1 O.E.
Tel: 45-78-06, 45-55-46
Cable: AARIS Bogota
Telex: 4440DINSTCO

COSTA RICA
Lic. Alfredo Gallegos Gurdián
Apartado 10159
San José
Tel: 21-86-13
Cable: **GALGUR San José**

ECUADOR
Laboratorios de Radio-Ingenieria
Calle Guayaquil 1246
Post Office Box 3199
Quito
Tel: 212-496; 219-185
Cable: HORVATH Quito

EL SALVADOR
Electronic Associates
Apartado Postal 1682
Centro Comercial Gigante
San Salvador, El Salvador C.A.
Paseo Escalon 4649-4° Piso
Tel: 23-44-60, 23-32-37
Cable: ELECAS

GUATEMALA
IPESA
Avenida La Reforma 3-48,
Zona 9
Guatemala
Tel: 63627, 64736
Telex: 4192 TELTRO GU

MEXICO
Hewlett-Packard Mexicana,
S.A. de C.V.
Torres Adalid No. 21, 11° Piso
Col. del Valle
Mexico 12, D.F.
Tel: 543-42-32
Telex: 017-74-507

NICARAGUA
Roberto Terán G.
Apartado Postal 689
Edificio Terán
Managua
Tel: 3451, 3452
Cable: ROTERAN Managua

PANAMA
Electrónico Balboa, S.A.
P.O. Box 4929
Ave. Manuel Espinosa No. 13-50
Bldg. Alina
Panama City
Tel: 230833
Telex: 3481103, Curunda,
Canal Zone
Cable: ELECTRON Panama City

PARAGUAY
Z. J. Melamed S.R.L.
Division: Aparatos y Equipos Medicos
Division: Aparatos y Equipos Scientificos y de Investigacion
P.O. Box 676
Chile, 482, Edificio Victoria
Asuncion
Tel: 4-5065, 4-6272
Cable: RAMEL

PERU
Compañia Electro Médica S.A.
Ave. Enrique Canaual 312
San Isidro
Casilla 1030
Lima
Tel: 22-3900
Cable: ELMEO Lima

PUERTO RICO
San Juan Electronics, Inc.
P.O. Box 5167
Ponce de Leon 154
Pda. 3-PTA de Tierra
San Juan 00906
Tel: (809) 725-3342, 722-3342
Cable: SATRONICS San Juan
Telex: SATRON 3450 332

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

VENEZUELA
Hewlett-Packard de Venezuela C.A.
Apartado 50933
Edificio Segre
Tercera Transversal
Los Ruices Norte
Caracas 107
Tel: 35-00-11
Telex: 21146 HEWPACK
Cable: HEWPACK Caracas

FOR AREAS NOT LISTED,

CONTACT:
Hewlett-Packard
Inter-Americas
3200 Hillview Ave.
Palo Alto, California 94304
Tel: (415) 493-1501
TWX: 910-373-1267
Cable: HEWPACK Palo Alto
Telex: 034-8300, 034-8493