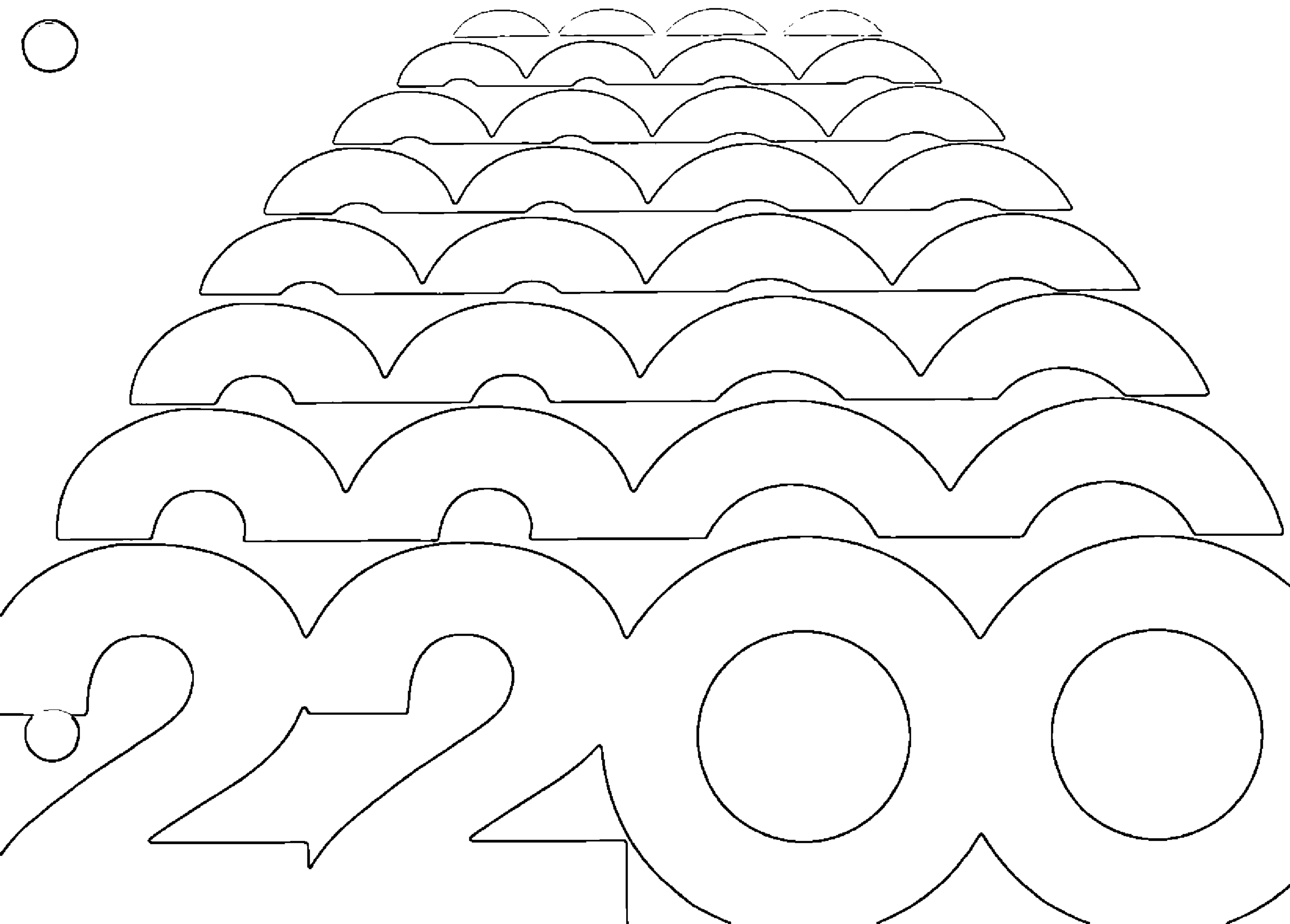
WANG

BASIC-2 DISK REFERENCE MANUAL



WANG BASIC-2 DISK REFERENCE MANUAL

1st Edition — June, 1981
Copyright © Wang Laboratories, Inc., 1981
700-4081G

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

HOW TO USE THIS MANUAL

The Wang BASIC-2 Disk Reference Manual is designed to serve as a programmer's guide to the concepts and features of disk utilization, and a reference guide for the BASIC-2 instructions which govern disk operations.

Chapter 1 introduces the concepts and features of information storage and retrieval on the disk and the general procedures for addressing and accessing a disk drive under program control. These procedures are common to all Wang disk models.

Wang System 2200 provides two modes of disk operation - Automatic File Cataloging Mode and Absolute Sector Addressing Mode. Chapters 2, 3, and 4 constitute a programmer's guide to the features available in Automatic File Cataloging Mode. Chapter 2 may be of particular interest to the beginning disk programmer, because it serves as a primer for disk operations, explaining fundamental concepts of disk management such as file structure, record layout, file and record accessing, etc.

Chapter 5 is a reference chapter for the BASIC-2 statements which comprise the Automatic File Cataloging Mode. The syntax and capabilities of each statement are presented in a brief, compact format which makes it quickly accessible to the programmer who is already familiar with the general principles of Automatic File Cataloging.

Chapter 6 serves as a programmer's guide for the Absolute Sector Addressing Mode. Chapter 7 is a companion reference chapter which describes the syntax and capabilities of the BASIC-2 instructions of Absolute Sector Addressing Mode.

Chapter 8 is a hybrid chapter incorporating both hardware and programming information on the disk multiplexer (Model 2230MXA-1/B-1), which permits a single disk unit to be accessed by several CPU's. Multiplexer owners should consult this chapter before attempting to install or program the multiplexer.

Explanations of the disk error codes, comparisons of BASIC and BASIC-2 disk instructions, a bibliography of disk literature, a glossary of disk terminology, and other information of interest to the disk user has been assembled in the Appendices.

TABLE OF CONTENTS

LIST OF EXAMPLES

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
ACCESSING A DISK PLATTER

1.1 INTRODUCTION

Information on a disk platter is stored on concentric circular tracks which are divided into a number of discrete segments called sectors. Each sector has a fixed storage capacity of 256 bytes and has its own sector address which allows it to be directly accessed by the system. The programmable instructions used to access the disk platters are essentially the same for all Wang disk models. The following sections discuss the procedures for accessing the disk platters when the system contains only one disk unit or when accessing the primary disk drive from among multiple disk units. The considerations for accessing the disk platters when the system contains more than one unit are discussed in Chapter 3.

1.2 THE PLATTER PARAMETERS

Before it can perform a disk read or write operation, the system must know which platter is to be accessed from a disk drive. Each disk platter is regarded by the system as an independent logical unit, with its sectors independently numbered starting at zero. In order to locate a given sector, the system must be told which drive unit and platter contain the desired sector. The drive is always specified by its device address. The platter is generally specified by the platter parameter, although the device address is sometimes used for platter specification.

Model 2260 and 2270 Series

The system uses the 'F' parameter and the 'R' parameter to uniquely identify the two platters in the Model 2260 series disk drives. The 'F' and 'R' parameters were designed originally for the fixed/removable disks, and they are mnemonics for "fixed" ('F') and "removable" ('R'). Thus, the 'F' parameter uniquely identifies the fixed platter, and the 'R' parameter identifies the removable platter (disk cartridge) in these disk units.

'F' and 'R' also may be used with diskette drives (Models 2270 and 2270A) and the minidiskette drives. All diskettes are equally removable; no diskette has a privileged status. 'F' and 'R' therefore have no particular mnemonic significance in this case. 'F' identifies the leftmost diskette drive (#1), while 'R' identifies the second drive (#2). In a Model 2270-1/2270A-1 or minidiskette configuration, with only one drive, 'F' identifies the single drive, and 'R' is not used. In a Model 2270-3/2270A-3, with three drives, 'F' serves a double duty, identifying both drive #1 and drive #3. When used to reference drive #3, however, the 'F' parameter must be accompanied by a special disk device address (see Section 1.5).

Table 1-1. Platter Parameters for the Model 2260 and 2270 Series

PARAMETER	MODELS	MODELS	MODELS	MODELS
	2260C 2260BC	2270-1 2270A-1 Minidiskette	2270-2 2270A-2 Dual Minidiskette	2270-3 2270A-3
F	Fixed Platter	Drive #1	Drive #1	Drive #1 or #3*
R	Removable Platter	Not Used	Drive #2	Drive #2

*F' parameter must be accompanied by special disk device address to access drive #3. See Section 1.5.

Model 2280 Series Disk Drive

The Model 2280 series disk drive contains a 13.4 megabyte removable platter and a fixed storage section, the size of which depends on the model available (i.e., 2280-1, 2280-2, or 2280-3). The fixed storage section of the disk is divided into a number of independent logical platters, each 13.4 megabytes in size. A particular platter (removable or fixed) is generally referenced by specifying the platter number as the low digit of the device address (see Table 1-2). The platter parameter 'T' is normally used for the Model 2280 series since it causes the system to use the platter designated by the device address. (See Section 3.7 for a full discussion of the 'T' parameter in disk operations.)

Table 1-2. **Platter Specifications for the Model 2280**

MODEL	PLATTER	DEVICE ADDRESS
2280-1	Removable	D10 (or B10)*
	Fixed	D11 (or 310)
2280-2	Removable	D10 (or B10)
	First Fixed	D11 (or 310)
	Second Fixed	D12
	Third Fixed	D13
2280-3	Removable	D10 (or B10)
	First Fixed	D11 (or 310)
	Second Fixed	D12
	Third Fixed	D13
	Fourth Fixed	D14
	Fifth Fixed	D15

*For compatibility with other disk units, the removable and the first fixed platter can be addressed in the same manner as the 2260 Series disk units. In this case the first character of the device address must be 3 or B.

1.3 ACCESSING A DISK PLATTER

Access to a particular platter is obtained by including the appropriate platter specification in a disk statement. For example, the Absolute Sector Addressing statement DATASAVE DA is used to store data on disk beginning at a specified sector address. For example, to record data in sector #100 on the fixed platter of a Model 2260 BC disk unit, or on the diskette mounted in drive #1 of a Model 2270-3 diskette unit, the following statement might be used:

```
10 DATASAVE DA F (100) A$
```

Similarly, the following statement could be used to record data in sector #100 on the removable platter, or on a diskette mounted in drive #2 of a diskette unit:

```
10 DATASAVE DA R (100) A$
```

Likewise, the following statements could be used to record data in the specified sectors on the removable and fixed platters of the Model 2280-2.

```
10 DATASAVE DA T/D10 (100) A$
20 DATASAVE DA T/D11 (100) B$
30 DATASAVE DA T/D12 (100) C$
40 DATASAVE DA T/D13 (100) D$
```

The Automatic File Cataloging procedures, discussed in Chapters 2-4, permit the programmer to read and write information on disk without specifying a sector location. In this case, the Catalog automatically keeps track of where each file is stored, so that the programmer only needs to provide the file name and the platter specification. For example, the SAVE statement is used to record named programs on disk with Catalog procedures. Thus, the following statement could be used to save the program PROG-1 on the fixed platter of a 2260 series disk drive, or on diskette #1 of a diskette drive unit:

```
10 SAVE F "PROG-1"
```

The same program could be saved on the removable platter, or on diskette #2 of a diskette unit, with the following statement:

```
10 SAVE R "PROG-1"
```

1.4 THE DISK DEVICE ADDRESS (MODEL 2260 and 2270 SERIES)

In addition to the platter parameters "F" and "R" , which identify individual disk platters within a disk unit, the disk unit as a whole is identified with a unique three-digit disk device address. The disk device address enables the system to distinguish the disk from other peripheral devices (printers, plotters, etc.) and from other disk units in the same system. The device address of the first or primary disk unit in a system is 310. If the system supports two or more disk units, the disk device address is incremented by HEX(10) for each additional disk unit. (For example, the address of a second disk unit on the same system is 320; the address of a third, 330, etc.) The device addressing scheme for multiple-disk systems is covered in greater detail in Chapter 3, Section 3.8. For example, the following statement would save a file named "PROG-1" on the platter designated "F" on the default disk unit (designated 310).

```
10 SAVE F /310, "PROG-1"
```

Notice that the device address is preceded by a slash ("/") when specified directly in a disk statement. The indirect specification of a disk address involves the use of file numbers, and discussion of this technique is postponed until Chapter 3, where file numbers are introduced.

NOTE:

The disk device address may be omitted from a disk statement if there is only one disk unit in the system, or if the primary disk unit in a multiple-disk system is to be accessed since the system assumes a device address of 310 automatically (unless the default address has been changed by SELECT). Thus, the following pair of statements are, in general, equivalent:

```
10 SAVE F /310, "PROG-1"
        or
10 SAVE F "PROG-1"
```

In either case, the disk unit with address 310 is accessed.

1.5 ACCESSING THE THIRD DRIVE (MODEL 2270-3 and 2270A-3) AND SLAVE DRIVE (MODELS 2260C-2, 2260BC-2 and 2280 DUAL DRIVE)

In general, the device address of drive #3 in a Model 2270-3 (or 2270A-3) is determined by ORing HEX(40) to the primary address assigned to drives #1 and #2. For example, if the primary address of the triple drive is 310, the address of drive #3 is 350; if the primary address is 320, the address of drive #3 is 360, etc.

If drive #3 is accessed with the 'F' parameter, the special address must be referenced. For example, statement 10 below would access drive #1 of a triple drive unit:

```
10 SAVE F "PROG-1"
```

Alternatively, statement 20 accesses drive #3:

```
20 SAVE F /350, "PROG-2"
```

These statements assume, of course, that the primary address of the triple drive is the default address, 310.

The Slave drive in a Model 2260C-2, 2260BC-2 or 2280 dual drive system is treated much the same as the third drive in the Model 2270-3. Although it is functionally an extension of the Master drive, it is assigned a separate address which is always HEX(40) greater than the address of the Master disk. If the Master drive's default address is 310, the address of the Slave drive would be 350; if the address of the Master drive is 320, the address of the Slave drive is 360, etc. As an example, for the 2260C-2 or 2260BC-2, statement 10 below accesses the fixed platter in the Slave drive assuming that the primary address in the Master drive is the default address, 310.

```
10 SAVE F/350, "FILE4"
```

The statement 10 below accesses the first fixed platter of the Slave drive in a 2280 dual drive system.

```
10 SAVE T/D51, "FILE6"
```

CHAPTER 2
AUTOMATIC FILE CATALOGING PROCEDURES

2.1 INTRODUCTION

Once a disk platter has been formatted by the procedures described in the appropriate Disk Users Manual, it is ready to receive programs and data. The system provides two modes of disk operation for recording information on disk: Automatic File Cataloging mode, and Absolute Sector Addressing mode.

In Automatic File Cataloging mode, the system provides a set of procedures which automatically keep track of the size and location of each cataloged file. The process of storing and retrieving information on the disk is greatly simplified, because the system automatically performs many of the complex "housekeeping" chores associated with the maintenance of disk files. A drawback of this mode of operation, however, is that the Automatic File Cataloging statements do not permit the programmer to directly access individual sectors on disk.

By contrast, Absolute Sector Addressing mode enables the programmer to directly access any sector on disk by specifying its sector address. In this mode, the locations of all files on disk must be maintained by the programmer himself in his controlling software. Absolute Sector Addressing statements can be used to design custom disk file maintenance systems, or to write special search and sort routines which may be used in conjunction with cataloged files.

Chapters 2, 3, 4, and 5 describe and explain the Automatic File Cataloging procedures. This chapter introduces the concept of cataloging, and discusses many of the most basic cataloging procedures, including storage and retrieval of programs and data on disk, skipping to particular records within a data file, listing the contents of the Catalog Index, and creating backup copies of cataloged files on a second platter. Chapters 3 and 4 cover these and other subjects in greater detail. Chapter 5 lists the general forms of all catalog statements, with each statement's parameters shown in a clear and readable form. The general forms are listed in alphabetic order for ease of reference. Absolute Sector Addressing procedures are dealt with in Chapters 6 and 7.

2.2 WHAT IS AUTOMATIC FILE CATALOGING?

Automatic File Cataloging mode consists of 18 BASIC statements, which invoke a set of built-in routines which perform specific file control functions. The catalog statements enable a programmer to create and access program files and data files on the disk by name, without reference to specific sector locations. Each newly created file is automatically placed in an available location by the system, and the file's name and location are recorded for future reference. In addition, a number of auxiliary file maintenance operations (such as skipping records within a file, creating backup copies of files, and providing essential file parameters) are supported.

Clearly, the most important function of the catalog procedures is to keep an accurate account of where each file is located on a disk platter. Such a function may not be critical when the platter contains only a single file, but if two or more files are stored on the same platter, it is imperative that accurate records of the size and location of each file be kept.

The structure used to keep track of where each file is stored is called the "catalog". The catalog consists of two sections, a "Catalog Index" and a "Catalog Area". All cataloged program files and data files are physically stored in the Catalog Area, which typically occupies the major portion of a platter's storage area. The Catalog Index, which normally occupies a much smaller portion of the platter, contains the name and location of each file. Files are stored sequentially in the Catalog Area, and their names and locations are recorded in the Catalog Index.

A file's "location" is expressed in terms of a "sector address". It was pointed out in an earlier chapter that the storage area of each platter is segmented into a number of storage blocks called sectors. Each sector has a total storage capacity of 256 bytes. The sectors on a platter are numbered sequentially starting at zero; the number assigned to each sector is referred to as its "address". Files typically occupy a number of sequential sectors, and the files themselves are always stored sequentially in the Catalog Area. For example, FILE#1 may occupy 50 sectors, starting at sector #100 and proceeding sequentially to sector #149. A second file, FILE#2, would automatically be stored beginning at sector #150. If FILE#2 also is 50 sectors in length, it occupies sectors #150 - #199. The location of a file is simply the address of the first sector in the file. Thus, the system might make the following entries in the Catalog Index for the two files, FILE#1 and FILE#2:

NAME	STARTING SECTOR ADDRESS	ENDING SECTOR ADDRESS
FILE#1	100	149
FILE#2	150	199

When the programmer wishes to access one of the existing files, FILE#1 or FILE#2, he need only provide the system with the desired file name. If he requests FILE#2, the system looks for the name "FILE#2" in the Catalog Index, and quickly finds the location of FILE#2 in the Catalog Area. Because the Catalog Index is updated by the system whenever a new file is stored, and consulted by the system whenever an existing file is accessed, the programmer does not need to be concerned about where a cataloged file is actually located on the platter.

2.3 SUMMARY: WHAT THE CATALOG PROCEDURES CAN AND CANNOT DO

Although the catalog procedures can perform a number of critical file maintenance functions, there are a great many maintenance operations they do not perform, and they should not be regarded as constituting a complete disk file maintenance system.

In summary, Automatic File Cataloging mode includes statements which perform the following services for the programmer:

1. Provide the capability to establish a catalog, consisting of a Catalog Index and Catalog Area, on a designated disk platter.

2. Provide the capability to save programs on disk by name, without reference to a sector location, and load programs from disk into memory by name, without referencing a sector location.

3. Provide the capability to open and re-open data files on disk by name, without reference to a sector location.

4. Provide the capability to store data in an open data file without reference to a sector location.
5. Automatically write multi-sector records on the disk when the argument list requires it.
6. Provide the capability to skip forward and backward over data records within a data file.
7. Provide a complete listing of the contents of the Catalog Index.
8. Provide the capability to scratch unwanted files on disk, and reuse the space occupied by such files.
9. Provide the capability to copy the entire contents of the catalog (Catalog Index and Catalog Area) onto a second disk platter.

Some of the more common file maintenance procedures not supported by Automatic File Cataloging are listed below:

1. Direct access to individual sectors is *not* supported.
2. Random access to data records within a cataloged file is not supported (sequential access only is supported).
3. Search and sort routines, which in general require a direct-access capability, are not supported.
4. Automatic blocking of multiple data records within a single sector is not supported. (This must be simulated with the user's software.)
5. Overlapping a file from one platter to another is not supported. (Files are assumed to reside entirely on a single platter.)

Note, however, that all of the above procedures can be carried out by software routines written with the Absolute Sector Addressing mode statements, which do permit direct access to sectors on disk. Absolute Sector Addressing statements can be used to construct a wholly customized maintenance system, and they can be used in conjunction with Automatic File Cataloging statements to augment the catalog procedures. The programmer should carefully analyze his own application to determine which mode, or combination of modes, is suitable for his needs. The novice programmer is warned, however, against attempting to "mix" modes before acquiring a complete understanding of the catalog procedures discussed in Chapters 2, 3, 4, and 5.

2.4 INITIALIZING THE CATALOG

Before any information can be recorded on the disk with Catalog procedures, the catalog itself must be set up or "initialized" with the SCRATCH DISK statement. SCRATCH DISK enables the programmer to establish a Catalog Index and Çatalog Area on a specified disk platter. If a Catalog Index and Catalog Area have previously been set up, the SCRATCH DISK statement will destroy the existing information. The following items of information are included in a SCRATCH DISK statement:

1. The disk platter on which the catalog is to be established. Separate and independent catalogs are established on each disk platter, and each must be initialized independently. One of the platter parameters 'F' 'R' or 'T' is used to specify the desired disk platter.

2. The number of sectors which are to be reserved for the Catalog Index (any number between 1 and 255 is allowed). The 'LS' parameter is used for this purpose. The Catalog Index automatically begins at sector #0 on a platter. Therefore, if 40 sectors are reserved, the Index occupies sectors 0 - 39.

3. The address of the last sector to be used for the Catalog Area. This is usually the highest sector address on the platter. (Cataloged files cannot be stored on the disk beyond this sector.) The 'END' parameter is used for this purpose. The Catalog Area automatically begins with the first sector following the Catalog Index. For example, if 40 sectors (0 - 39) are reserved for the Index, the Catalog Area automatically begins at sector #40. The beginning point is a system function, and cannot be altered by the programmer. The address ot the last sector of the Catalog Area must, however, be specified. It must be less than or equal to the highest (last) sector address on the platter.

Example 2-1: Initializing the Catalog (2260, 2270 Series and Minidiskette)

```
10 SCRATCH DISK F LS=20, END=1000
```

Statement 10 instructs the system to initialize a catalog on the disk platter designated by 'F' ('F' designates the fixed disk platter on a Model 2260 series disk unit, and Drive #1 on the Minidiskette and Model 2270/2270A). Twenty sectors are reserved for the Catalog Index on this platter (LS = 20), and sector 1000 is specified as the last sector in the Catalog Area (END = 1000). Note that each disk platter must be initialized separately (i.e., with a separate SCRATCH DISK statement).

Example 2-2: Initializing a Catalog on a Model 2280 Disk Platter

```
10 SCRATCH DISK T/D12,LS = 40, END = 9000
```

Statement 10 instructs the system to initialize a catalog on the fixed platter designated by D12. Forty sectors are reserved for the Catalog Index on the platter (LS = 40), and sector 9000 is specified as the last sector in the Catalog Area (END = 9000).

In deciding how many sectors you should allocate for the Catalog Index, keep in mind that the first sector of the Index (sector 0) can hold 15 file names, and each subsequent sector (up to sector 254) can hold 16 file names. Thus, if you intend to store 15 or fewer files on a disk platter, one sector will be adequate for the Index. If you intend to store 16 or more files, two or more sectors must be reserved for the Index. It is important to note, however, that the size of the Catalog Index, once fixed, cannot be altered without reorganizing the entire catalog. The Index should therefore generally be allotted enough space to provide for any possible future additional files.

It is not strictly necessary to specify the number of sectors to be reserved for the Catalog Index. If you do not specify the number of sectors to be reserved in your SCRATCH DISK statement (i.e., if the 'LS' parameter is omitted), the system automatically reserves the first 24 sectors on the disk platter for a Catalog Index. Since the first sector of the Catalog Index (sector 0) holds a maximum of 15 file names (and associated file information) and each subsequent sector holds a maximum of 16 file names, a Catalog Index of 24 sectors provides enough space to store a maximum of 383 file names.

Example 2-3: Initializing the Catalog ('LS' Parameter Omitted)

```
20 SCRATCH DISK R END = 1000
```

Statement 20 instructs the system to establish a Catalog on the disk platter designated by 'R'. Sector 1000 is specified as the last sector to be used in the Catalog Area (END = 1000). Since the 'LS' parameter is omitted, the system automatically reserves the first 24 sectors on the 'R' platter for the Catalog Index.

Example 2-4: Initializing a Catalog on the Model 2280 ('LS' Parameter Omitted)

```
20 SCRATCH DISK T/D10, END = 10000
```

Statement 20 instructs the system to establish a Catalog on the removable disk platter (D10). Sector 10000 is specified as the last sector to be used in the Catalog Area (END = 10000). Since the 'LS' parameter is omitted, the system automatically reserves the first 24 sectors on the removable platter for the Catalog Index.

2.5 THE 'DC' PARAMETER

Certain BASIC verbs (e.g., SAVE, LOAD, DATASAVE, DATALOAD) are used in both the Automatic File Cataloging mode and the Absolute Sector Addressing mode. A special parameter is included in the BASIC statement to inform the system which mode of disk operation is meant. For catalog operations, the 'DC' parameter (mnemonic for "Disk Catalog") is used. Thus, for example, the statement SAVE DC is always interpreted by the system as a catalog statement (the remaining parameters in the statement must, of course, be correct). Absolute Sector Addressing mode is signalled by the 'DA' parameter ("Direct Addressing"). Thus, the SAVE DA statement always refers to Absolute Sector Addressing mode. A third parameter, 'BA' is used in two special statements which constitute a subclass of the 'DA' statements. The 'BA' statements are discussed, along with the 'DA' statements, in Chapter 6, "Absolute Sector Addressing".

Earlier Wang 2200 systems supported both disk and tape cassette operations, and tape was considered the default storage device. Thus, statements such as SAVE, LOAD, DATASAVE, and DATALOAD were regarded as specifying tape operations unless one of the disk parameters 'DC' 'DA' or 'BA' were included in the statement. The System 2200VP/MVP, however, does not support tape cassette operations. The disk itself is the default storage device, and catalog mode is the default mode of disk operation. In general, therefore, the 'DC' parameter is required in a disk statement. There are two disk statements that allow the optional use of the 'DC' parameter - the SAVE statement and the LOAD statement. Thus, the statement

```
SAVE F "PROG#1"
```

is equivalent to the statement

```
SAVE DC F "PROG#1"
```

Both statements cause PROG#1 to be saved on the 'F' platter under catalog procedures.

Likewise, for Model 2280 addressing, the statement

```
SAVE T/D15, "PROB4"
```

is equivalent to the statement

```
SAVE DC T/D15, "PROB4".
```

Both statements cause PROB4 to be saved on the fixed platter designated by D15.

Retention of the 'DC' parameter specifying disk operations permits compatibility with existing 2200 software, since 'DC' is required in Wang BASIC disk catalog statements. The following disk statements require the 'DC' parameter:

```
DATA LOAD DC
DATA LOAD DC OPEN
DATA SAVE DC
DATA SAVE DC OPEN
DATA SAVE DC CLOSE
LIST DC
```

2.6 SAVING CATALOGED PROGRAMS ON DISK: THE "SAVE" STATEMENT

Once the catalog is initialized, you can begin storing cataloged information on the disk. In catalog mode, all information must be stored in a named file on the disk. Cataloged disk files may be of two types: program files and data files. A data file may contain a large collection of data. A program file, however, always contains only one BASIC program or program segment.

Program files are stored on the disk with the SAVE statement. Each time a SAVE statement is executed, it creates a single named program file on disk. A program file consists of the BASIC program or program segment being saved, as well as certain file control information automatically included in the file by the system when the program is stored on disk.

In order to record a cataloged program on disk, the following information must be specified in the SAVE statement:

1. The disk platter on which the program is to be stored. The specified disk platter must have been initialized with a SCRATCH DISK statement.
2. The name of the program. You must name the program so that the system has some way of identifying it when it is stored on the disk. The name can be from one to eight characters in length. It may be specified as a literal string, or as the value of an alphanumeric variable. Each program must be given a unique name.

Example 2-5: Saving a Program on Disk

```
SAVE R "PROG#1"
```

This statement instructs the system to transfer all program lines currently in memory to the default drive disk platter designated by 'R' and name this program file "PROG#1". The file's name ("PROG#1") and location are automatically listed in the Catalog Index.

Example 2-6: Saving a Program on a Fixed Disk of the Model 2280

```
SAVE T/D12, "PROB#7"
```

This statement instructs the system to transfer all program lines currently in memory to the fixed platter designated by D12 and names this program file "PROB #7".

It is also possible to save just a portion of a program currently in memory. This is accomplished by including the appropriate line numbers in the SAVE statement (Examples 2-7 and 2-8).

Example 2-7: Saving Part of a Program on Disk (One Line Number Specified)

```
SAVE R "PROG#2" 200
```

This statement instructs the system to transfer all statement lines in memory beginning with line 200 through the highest numbered line onto the disk platter designated by 'R'. The program is named "PROG#2" and its name and location are automatically entered in the Catalog Index.

Example 2-8: Saving Part of a Program on Disk (Two Line Numbers Specified)

```
30 A$ = "PROG#3"
40 SAVE R A$ 200, 500
```

This statement transfers program lines 200 through 500 from memory to the 'R' disk platter. The program is named "PROG#3" since that is the value of A$, and the program's name ("PROG#3") and location are entered in the Catalog Index.

A useful feature of the SAVE statement is the ability to delete unnecessary spaces and remarks (REM statements) in a program before it is stored on the disk. If the parameter <S> is specified in the SAVE statement, all unnecessary spaces (excluding spaces in character strings enclosed in quotes or in % statements) will be deleted from the program as it is saved. By specifying the parameter <SR> in the SAVE statement, both unnecessary spaces and remarks are removed from the program when it is saved on disk.

Example 2-9: Saving a Program on Disk with SAVE Using the <S> Parameter

```
10 REM ENTER VARIABLES IN PROGRAM
20 A=   5.64:B=4.8
30 C$ = "SALES TAX"
        .
        .
        .
100 PRINTUSING 160
        .
        .
        .
160 % ACCT. NO.      AMOUNT
SAVE <S> F (400) "ACCOUNTS"
```

In example 2-9, SAVE transfers the program "ACCOUNTS" from memory to the 'F' disk platter. The parameter <S> deletes the unnecessary spaces in line 20 of the program when it is saved. The spaces enclosed in quotes in line 30 and the spaces following the % statement in line 160 are not deleted. If the parameter <SR> is used in SAVE instead of <S>, then line 10 is also deleted when the program is saved on disk.

2.7 RETRIEVING PROGRAMS STORED ON DISK: THE "LOAD" INSTRUCTION

Cataloged programs are retrieved from the disk with the LOAD instruction. LOAD is referred to as an "instruction" rather than a "statement" or a "command" because it is a hybrid, functioning differently depending upon its mode of execution. When executed in Immediate Mode, LOAD is referred to as the LOAD command, and stimulates a specific sequence of operations. When executed in program mode (i.e., on a numbered program line), it is referred to as the LOAD statement; in this case, an altogether different sequence of operations are initiated. Because of the operational distinctions between the two forms of LOAD, the LOAD statement and the LOAD command are discussed in separate sub-sections.

The LOAD Command

The LOAD command is never executable in program mode; it is executed in immediate mode only. If the LOAD instruction appears in a program (i.e., on a numbered program line), it is interpreted as a LOAD statement, and the operations associated with the LOAD statement are carried out by the system. The LOAD command instructs the system to locate a named program on a specified disk platter, and load that program into memory. The system first checks the Catalog Index for the specified program name, and then, upon locating the name, determines the program's location in the Catalog Area, and moves to that location to load the program.

Following execution of the LOAD command, the newly loaded program is appended to existing program text in memory. New program lines which have the same numbers as program lines already stored in memory replace the currently stored lines in memory. Currently stored program lines which do not have the same line numbers as new program lines are not cleared, however; they remain as lines in the new program. (For example, if the old program has lines numbered 5, 15, 25, etc., and the newly-loaded program lines are numbered 10, 20, 30, etc., the new program in memory has lines numbered 5, 10, 15, 20, etc.) For this reason, it is generally wise to clear memory prior to loading the new program. All of memory can be cleared by executing a CLEAR command prior to executing the LOAD command. Alternatively, a CLEAR P command causes only program text to be cleared from memory. After the new program is loaded, it is necessary to key RUN and (EXEC) in order to execute the newly loaded program.

The LOAD command must include the following two items of information:

1. The disk platter parameter ('F' 'R' or 'T') on which the desired program is stored.
2. The name of the program which is to be retrieved (the name may be specified as a literal string, or as the value of an alphanumeric variable).

Example 2-10: Loading a Cataloged Program File from Disk with the LOAD Command

```
CLEAR
LOAD R "PROG#1"
```

This command instructs the system to load PROG#1 into memory from the disk platter designated by 'R'. When the command is executed, the system accesses the 'R' platter and searches for the program name "PROG#1" in the Catalog Index. Upon locating the name in the Catalog Index, the system checks the starting sector address of the program, and moves to that address in the Catalog Area to begin loading PROG#1 into memory. Note that if CLEAR is not executed the new program is appended to existing program text in memory (new program lines which have the same number as program lines already in memory replace the resident lines in memory). After the program is loaded, it is necessary to key RUN and (EXEC) in order to begin execution of the new program.

Example 2-11: Loading a Cataloged Program File from a Model 2280 Disk Platter using the LOAD Command

```
CLEAR
LOAD T/D10 "PROB#7"
```

This LOAD command instructs the system to load PROB#7 into memory from the removable disk platter designated by D10. Before the program is loaded, all memory is cleared with the CLEAR command.

If the program name specified in a LOAD command is not located in the Catalog Index on the specified disk platter, an error is indicated. Note also that the program name supplied in a LOAD command must correspond exactly to the program name listed in the Catalog Index. Any misspelling results in an error.

Example 2-12: Attempting to Load a Non-Cataloged Program from Disk

```
CLEAR
LOAD R "PRAG#1"
```

This command is meant to retrieve PROG 1 from the 'R' disk platter. Because the program's name is misspelled, however ("PRAG#1" instead of "PROG#1"), the system cannot find a program under this name in the Catalog Index. It therefore signals an error:

```
LOAD R "PRAG#1"
              ↑ERR D82
```

Where Error D82 = "File Not in Catalog".

The LOAD Statement

Cataloged programs can also be loaded into memory from disk under program control. The LOAD statement is used for this purpose. The LOAD statement is executable only in a program (i.e., on a numbered program line). When the LOAD instruction is executed in immediate mode, it is always interpreted as a LOAD command, and the sequence of operations associated with the LOAD command (see above) is followed by the system.

The following sequence of operations is associated with the LOAD statement:

1. Stop current program execution.
2. Clear all currently stored program text (or a specified portion of currently stored program text) from memory.
3. Clear all noncommon variables from memory.
4. Locate the named program on the specified platter. (If the specified name cannot be found in the Catalog Index, an error is signalled.)
5. Load the program into memory.
6. Run the newly loaded program.

In a LOAD statement, the system must be provided with the following information, in the order indicated:

1. The disk platter on which the desired program is stored.

2. The name of the program which is to be loaded (the name may be specified as a literal string in quotes, or as the value of an alphanumeric variable).

Optionally, a third item may be included:

3. One or two program line numbers which specify the first and last program lines to be cleared from memory prior to loading the new program.

In addition, several programs may be loaded at once from the disk:

4. An expression enclosed within < > specifying the number of files to be loaded, and an alpha-variable to list the names of the programs to be loaded.

If no line number is specified in the LOAD statement, the system clears all program text from memory prior to loading the new program from disk. As soon as the program is loaded, execution begins automatically at the first (lowest) program line in the newly loaded program. The LOAD statement is commonly used to "chain" programs from the disk. Common variables (so specified in a COM statement) are retained in memory for use by each succeeding program in the chain. Noncommon variables are cleared by the LOAD statement.

Example 2-13: Chaining a Program from Disk with the LOAD Statement

```
100 LOAD F "PARTA#2"
```

When executed, statement 100 stops program execution, clears all program text and noncommon variables from memory, and loads in the program PARTA#2 from the 'F' disk platter. Execution of PARTA#2 begins automatically at the first (lowest) line in the program.

Example 2-14: Chaining a Program from a Model 2280 Disk Platter with the LOAD Statement

```
200 LOAD T/D14,"PARTA#4"
```

When executed, statement 200 stops program execution, clears all program text and noncommon variables from memory, and loads in the program PARTA#4 from the fixed platter designated D14. Execution of PARTA#4 begins automatically at the first (lowest) line in the program.

If program segments are to be overlayed from disk, it may be desirable to clear out only a specific portion of program text prior to loading the new program segment. In this case, one or two program line numbers can be included in the LOAD statement. Inclusion of a single line number in the statement causes all program text beginning at that line to be cleared from memory prior to loading the new program. Two line numbers instruct the system to clear all program text between and including the specified lines prior to loading the new program. In either case, all non-common variables are cleared. Execution of the newly loaded program begins at the first line number specified in the LOAD statement. If this line number does not appear in the newly loaded program, an ERROR P36 (Undefined Line Number) is signalled.

Example 2-15: Loading a Program Overlay from Disk with the LOAD Statement

```
200 LOAD F "PARTA#3" 300, 900
```

Statement 200 halts program execution and clears program lines 300 through 900 from memory, along with all non-common variables, prior to loading program overlay PARTA#3 from the 'F' platter. After PARTA#3 is loaded, program execution continues automatically at line 300. If PARTA#3 contains no line number 300, an ERROR P36 (Undefined Line Number) is signalled.

Several programs can be overlayed from the disk with a single LOAD statement. In this case, the LOAD statement must include the total number of programs to be loaded from the disk. The names of the programs are specified sequentially in an alpha-variable. (The alpha-variable must be a common variable.)

Example 2-16: Loading Several Cataloged Program Files from Disk with the LOAD Statement

```
10  COM B$(4)8
20  B$(1)="PARTA#1": B$(2)="PARTA#2"
    :B$(3)="PARTA#8": B$(4)="DOLLARS2"
30  LOAD R<4> B$()
```

In statement 10, B$(4) is designated a common variable consisting of four elements. Each element consists of a literal string with a maximum length of 8 bytes. Statement 20 defines the program names that are stored sequentially in B$(4). When executed, statement 30 stops program execution, clears all program text and noncommon variables from memory, and loads into memory from the 'R' disk platter the programs PARTA#1, PARTA#2, PARTA#8, and DOLLARS2 in that order. Execution begins at the lowest numbered line.

2.8 LISTING THE CATALOG INDEX: THE "LIST DC" STATEMENT

It is frequently helpful to know the names of all files stored on a particular platter. The names of all cataloged files on a platter are recorded in the Catalog Index, along with the location of each file, file type, status and total sectors reserved for the file. A list of the entries in a Catalog Index can be obtained with the LIST DC statement. Note that the 'DC' parameter is required in the LIST DC statement; if it is omitted, the system lists the resident program in memory rather than the Catalog Index. The platter parameter of the platter containing the index to be listed must also be included. When the LIST DC statement is executed, the following information is returned:

1. Information on the status of the catalog itself:

   a) The number of sectors reserved for the Catalog Index on the specified disk platter.

   b) The address of the last sector reserved for the Catalog Area.

   c) The current end of the Catalog Area (i.e., the last sector of the Catalog Area currently occupied by a catalog file).

2. Information on the status of each cataloged file:

   d) The name of each cataloged file.

   e) The file type (program or data) of each file, and its status ("S" if scratched).

   f) The starting and ending sector addresses of each file.

   g) The number of sectors used in each file.

   h) The number of free sectors available (not used) in each file.

Example 2-17: Listing the Catalog Index

```
50 LIST DC R
```

Statement 50 causes the system to list the contents of the Catalog Index from the 'R' disk platter. The 'R' platter was initialized in Example 2-3, and program files were written into the Catalog in Examples 2-5, 2-7, and 2-8. The listing therefore looks like this:

```
INDEX SECTORS = 00024
END CAT. AREA = 01000
CURRENT  END  = 00132

NAME          TYPE   START      END       USED     FREE

PROG#2        P      00051      00112     00062    00000
PROG#3        P      00113      00132     00020    00000
PROG#1        P      00024      00050     00027    00000
```

Figure 2-1. The Catalog Index Listing

There are several things which should be noticed about the information in this listing. Notice, first, that all files are stored sequentially, in the order in which they were saved in the Catalog Area. The Catalog Index occupies the first 24 sectors (sectors 0-23). The first file, PROG#1, is stored in the first sector of the Catalog Area, the first available sector following the Index (sector 24). PROG#2 begins at the first available sector following PROG#1 (sector 51), and PROG#3 starts with the first sector after PROG#2 (sector 113). Notice also, however, that the Catalog Index entries themselves are not listed in sequential order. That is because entries in the Catalog Index are stored in a "hashed" order, that minimizes the system's search time for locating entries in the Index. (A disk utility program is available that produces a listing in alphabetical order.)

You should observe, finally, that the USED column opposite each program name indicates the number of sectors used by that program. In the cases so far discussed, the system automatically used exactly enough sectors on the disk to store each program. Thus, notice that there are no unused sectors listed under the FREE column for any of the programs. It is possible, however, to reserve a number of sectors on the disk in addition to the number needed to store a program; the extra sectors can be used subsequently for additions to the program. The technique for reserving extra sectors for a program file is discussed in Chapter 4.

Displaying the Catalog Index in Steps with LIST S DC

If the Catalog Index contains a large number of entries, only the last entries can be conveniently read when the Index is displayed on the CRT (the preceding entries will flash by on the screen too quickly to be read). In this case, the Catalog Index listing can be displayed in sections by including the 'S' parameter in a LIST DC statement. The 'S' parameter is a somewhat unique parameter in that it precedes 'DC' in the LIST DC statement (e.g., LIST S DC). The statement

```
LIST S DC R
```

causes the 'R' platter Index listing to stop when the CRT screen is full. Likewise, the statement

```
LIST S DC T/D11
```

causes the Index Listing on the D11 platter (Model 2280) to stop when the CRT screen is full.

In order to display the next section of the listing, the operator must key RETURN(EXEC). Note that the LIST S DC statement is functionally similar to the LIST S statement, which displays the resident program in sections.

2.9 AN OPTIONAL METHOD OF LOADING CATALOGED PROGRAMS FROM DISK: THE "LOAD RUN" COMMAND

The LOAD RUN command provides a third method of loading cataloged programs from disk, in addition to the LOAD statement and LOAD command. Normally, RUN must be used to initiate execution of a program entered from the keyboard, or loaded from disk with the LOAD command. The LOAD RUN command, however, loads and immediately executes a program stored on disk. LOAD RUN provides a convenient means for the user to load and execute a menu program which can direct him to the desired program.

The LOAD RUN command in effect produces an automatic combination of the following operations:

1. CLEAR from memory all program text (including both common and non-common variables).
2. LOAD the named program from the designated platter.
3. RUN the program.

Example 2-18: Loading and Running a Cataloged Program with the LOAD RUN Command

```
LOAD RUN R "PROG#2"
```

This command causes the cataloged program PROG #2 to be loaded from the 'R' platter, and automatically run. Prior to loading, memory is cleared.

In example 2-18 if neither the platter nor the disk address are specified in the LOAD RUN command, the 'T' platter parameter and the 310 disk address are assumed. ('T' indicates that the platter is to be determined from the device address.) In addition, if no program name is specified, the system recognizes the default file name "START" (since many initialization programs utilize the name "START").

Example 2-19: Using the LOAD RUN Command with the Model 2280

```
LOAD RUN T/D13,"PROB#7"
```

This command causes the cataloged program PROB#7 to be loaded from the fixed platter designated by the address D13, and automatically run.

Example 2-20: Loading and Running a Cataloged Program with the LOAD RUN Command (Program "START" on Disk)

```
LOAD RUN
```

This command causes the cataloged program START to be loaded from the 'T' platter, and automatically run. If program START is not in the Catalog Index, error D82 (File Not In Catalog) is signalled.

2.10 SAVING DATA FILES ON DISK

The Hierarchy of Data

Unlike a program file, which always contains only a single program or program segment, a data file normally contains many different items of data. Obviously, it would be unwise to simply dump data on the disk in a random or disorganized fashion since there would then be no efficient way to retrieve specific items when they were needed. In order to facilitate fast, efficient retrieval of data from the disk, data stored on disk is organized into a well-defined structure or hierarchy.

The hierarchy of data is organized in the following way: items of data relating to a single subject (such as a particular customer, or a particular item in the inventory) are organized into a data record (also known as a logical record); a number of related data records are then stored in a data file (in this case, an inventory file or customer file). An inventory file, for example, would contain a number of inventory records, each of which contains information about an individual item in the inventory (such as model number, name, price, number in stock, etc.). Whenever a particular piece of information about one of the items in the inventory is needed, the procedure is first to locate the inventory file, and then to read the desired record from the file.

A number of different files can be established on disk or, less commonly, a single large file which occupies the entire Catalog Area. Within each file, the individual records can be as long as necessary (but each record must have a minimum length of at least one sector, unless special techniques are used to block records in a sector). In catalog mode, the system automatically keeps track of where each file is located on the disk. Usually, it is up to the programmer, however, to keep track of the locations of records within the file by using appropriate DSKIP and DBACKSPACE instructions (see Section 2.16).

Because the system itself has no way of knowing how many records will be stored in a file, or how long those records will be, it is up to the programmer to estimate how many sectors each file will require. The system must then be instructed to reserve adequate space for the file on a disk platter. Thus, two steps are required to save data on the disk:

1. First, the data file must be cataloged, or "opened" with a special statement, DATASAVE DC OPEN. In this statement, the new data file is named, and the number of sectors to be reserved for the file are specified. No data is actually stored in the file at this point.

2. Once the file is opened, data records can be stored in the file with the DATASAVE DC statement.

2.11 OPENING A CATALOGED DATA FILE ON DISK: THE "DATASAVE DC OPEN" STATEMENT

A data file is opened on the disk with a DATASAVE DC OPEN statement. In the statement, the following information must be provided:

1. The disk platter on which the data file is to be opened. This disk platter must have been initialized with a SCRATCH DISK statement (see Section 2.4).
2. The maximum number of sectors to be reserved for the data file. Take care that the file does not extend beyond the limits of the Catalog Area.
3. The name of the data file. The file must be given a unique name of one to eight characters in length, so that the system has some way of identifying it (embedded spaces in the name count as characters). The name may be specified either as a literal string or as the value of an alphanumeric variable.

When the DATASAVE DC OPEN statement is executed, the specified number of sectors are reserved for the newly-opened file in the Catalog Area on the designated platter. The last sector of the file is used by the system for a special control record which marks the absolute end of the file. No data can be written in the file beyond that point. The file's name and location are also automatically entered in the Catalog Index.

Example 2-21: Opening a Data File on Disk

```
150 DATASAVE DC OPEN F (100) "DATFIL-1"
```

Statement 150 instructs the system to reserve 100 sectors on the disk platter designated by 'F' and name this file "DATFIL-1". The file's name ("DATFIL-1") and location are entered automatically in the Catalog Index on the 'F' platter.

Example 2-22: Opening a Data File on a Model 2280 Disk Platter

```
100 SELECT #4/D11

200 DATASAVE DC OPEN T #4, (100) "FILE-2"
```

Statement 150 assigns the fixed platter address D11 to the #4 slot in the Device Table (See Section 3.2). This procedure allows the data file "FILE-2" to be opened on the D11 platter via statement 200. Statement 200 instructs the system to reserve 100 sectors on the D11 platter, and name this file "FILE-2". The file's name ("FILE-2") and location are entered automatically in the Catalog Index of the D11 platter.

NOTE:

The system automatically allocates the last sector in each data file exclusively for the system control record. The system control record contains control information and pointers used by the system in maintaining the data file, and no data should be stored in this sector. It is also generally desirable to write an end-of-file trailer record in a data file after all data has been stored; the trailer record likewise occupies one sector which cannot be used for data. Thus, it is always good programming practice to reserve at least two more sectors than are actually required for a data file in order to account for the two sectors used for control information and end-of file. For example, if you wish to store 24 sectors of data in a file, you should reserve at least 26 sectors (24 + 2) in the DATASAVE DC OPEN statement.

If you executed a LIST DC statement following statement 150 in Example 2-21, the listing should look like this:

```
INDEX SECTORS = 00100
END CAT. AREA  = 01000
CURRENT  END  = 00199

NAME          TYPE   START      END        USED       FREE

DATFIL-1      D      00100      00199      00001      00099
```

Figure 2-2. Catalog Index Entry for DATFIL-1

One hundred sectors are reserved for DATFIL-1 (00100-00199), but despite the fact that no data has yet been saved in the file, the USED column for DATFIL 1 indicates that one sector is already occupied, and the FREE column indicates that only 99 sectors are available for use. The last sector in the file, automatically set aside for system control information in DATFIL-1, cannot be used for data storage. Thus, although 100 sectors were reserved for DATFIL-1, only 99 of those sectors can actually be used for data storage. If 100 sectors are needed for data, at least 101 must be reserved for the data file.

NOTE:

Wang 2200 systems do not distinguish between input files and output files in disk operations. Thus, data can be either written in or read from a file which has been opened with a DATASAVE DC OPEN statement.

2.12 SAVING DATA IN A CATALOGED DATA FILE ON DISK: THE "DATASAVE DC" STATEMENT

Once a data file has been opened on a disk platter, it is an easy matter to store data in that file with a DATASAVE DC statement. In the DATASAVE DC statement, you must indicate only the data, or the alpha or numeric variables or arrays containing the data, which is to be saved. Individual items must be separated by commas. This information is known as the DATASAVE DC argument list. The system automatically groups information from the argument list sequentially in a logical data record, and stores this record sequentially in the currently open file on the disk.

Suppose, for example, that a group of 60 test results have been generated and must be stored in DATFIL-1. Since DATFIL-1 was recently opened (Example 2-21), it is the currently open data file on disk. Assuming that the 60 values are stored in an array A( ) which is dimensioned to contain 60 elements (i.e., DIM A (60)), the 60 values are written into DATFIL-1 simply by including the array name followed by closed parentheses (e.g., A( )) in the argument list of a DATASAVE DC statement, as in Example 2-23:

Example 2-23: Saving Data in a Data File

```
160 DATASAVE DC A( )
```

Statement 160 instructs the system to transfer all values from numeric array A( ) into the currently open data file on disk. Since, in this example, A( ) contains 60 numbers, and DATFIL-1 is the currently open file, 60 numeric values are transferred from A( ) into DATFIL-1. Elements from the array are transferred row by row and stored sequentially on the disk. Collectively, the 60 numeric values constitute one logical record on the disk.

If, after some data has been stored in DATFIL-1, a LIST DC F statement is executed, the Index looks like this:

```
INDEX SECTORS = 00100
END CAT. AREA = 01000
CURRENT  END  = 00199

NAME          TYPE   START      END        USED      FREE

DATFIL-1      D      00100      00199      00001     00099
```

Figure 2-3. Catalog Index Entry for DATFIL-1

Notice that the USED and FREE columns have not yet been updated to reflect the newly stored data in DATFIL-1. Since 28 full-precision numbers can be stored in one 256-byte sector, 60 such numbers occupy only two entire sectors and a portion of a third. Thus, the USED column for DATFIL-1 should read 00004, indicating that three sectors in DATFIL-1 have been used for data, in addition to the single sector reserved for system information. Similarly, the FREE column should read 00096. Why is this not the case?

The answer: The USED and FREE columns are updated only when an end-of-file record has been written to the file. The end-of-file (or trailer) record tells the system, in effect, "No data is stored in this file beyond this point." With this information, the system can determine how many sectors in the file are filled with data, and can update the USED and FREE parameters appropriately. A trailer record is not written to the file automatically, however. It must be created by the programmer with a DATASAVE DC END statement. The parameters for DATFIL-1 could be updated by following statement 160 in Example 2-23 with a DATASAVE END statement, as shown in Example 2-24:

Example 2-24: Writing an End-Of-File (Trailer) Record to a Cataloged Data File on Disk with DATASAVE DC END

```
170 DATASAVE DC END
```

Statement 170 instructs the system to write a trailer record into DATFIL-1.

If you now perform a listing of the Catalog Index, the Index looks like this:

```
INDEX SECTORS = 00100
END CAT. AREA = 01000
CURRENT END   = 00199

NAME        TYPE   START      END       USED     FREE
DATFIL-1    D      00100      00199     00005    00095
```

Updated USED now shows one sector used for file control, one sector for end-of-file record, and three sectors for data record.

Figure 2-4. Updated Catalog Index Entry for DATFIL-1

As you can see, the USED and FREE columns are now updated. It is good programming procedure to write a trailer record following every addition of new records to the file, so that it will always be possible to tell how much of the file is filled, and how much space remains. However, it is not necessary to write a trailer record after every DATASAVE DC statement; instead, a single DATASAVE DC END statement can be used at the conclusion of file update routine.

Example 2-25: Writing a Data Trailer Record after a Series of DATASAVE DC Statements

```
180 DATASAVE DC A()
190 DATASAVE DC B(), N,M(3)
200 DATASAVE DC A$,T$()
300 DATASAVE DC END
```

Statements 180-200 instruct the system to transfer data from the numeric and alphanumeric variables, arrays, and array elements specified in the respective argument lists, and store this data in the currently open data file (DATFIL-1) on disk. Statement 300 instructs the system to write an end-of-file trailer record following the last data record in DATFIL-1, and update the USED and FREE parameters for DATFIL-1 in the Catalog Index to indicate how many sectors have been used and how many are still available.

In addition to updating the USED and FREE parameters for a file, there are three major advantages to writing an end-of-file trailer record in a data file:

1. The trailer record makes it possible to skip to the end of stored data in a file in order to write new records into the file. (See Section 2.17.)
2. The trailer record makes it possible to test for the end of stored data (last record) in a file when reading through the file sequentially under program control. The IF END THEN statement is used for this purpose. (See Section 2.18.)
3. The trailer record insures against accidentally reading beyond the last valid data record in a file.

WARNING:

Never use the RESET button to terminate program execution during a disk write routine. RESET causes the disk to immediately terminate any operation and return the read/write head to the home position, even if it is in the middle of writing a sector. Thus, it is possible that a half-written sector may be left in the file following a RESET operation. Any subsequent attempt to read the half-sector results in an error. To avoid this problem, always use the HALT/STEP key if you wish to halt program execution during a disk write routine. HALT/STEP permits the disk to complete the write operation for the current statement before terminating the data transfer.

2.13 THE STRUCTURE OF DATA FILES

The discussion up to now has focused primarily on the mechanics of saving data on the disk; little attention has been paid to the actual manner in which data is organized and stored by the system. It will be helpful to consider this question briefly now (a more detailed discussion is reserved for the following chapter), prior to discussing the retrieval of data from a cataloged data file on disk.

The major concept to be understood in connection with data files is that of a logical data record. A single logical record (or data record) is created in a file on the disk with each DATASAVE DC statement. The logical record contains all of the data included in the DATASAVE DC argument list. For example, the following statements might be used to open a cataloged data file and store one logical data record in it:

```
10 DATASAVE DC OPEN F (200) "DATFIL-1"
20 DATASAVE DC "PETER RABBIT" 01121,B$,N,A()
```

Statement 10, as you know, opens DATFIL-1 on the F platter. Statement 20 creates one logical record in DATFIL-1 containing all the data from the DATASAVE DC argument list. Notice that there are several different types of data in the argument list. The first item is a literal string "PETER RABBIT". (Whenever a literal string is specified in a DATASAVE DC argument list, it must be enclosed in quotes.) The second item, 01121, is a numeric value which need not be set in quotes. The third item, B$, is an alphanumeric variable. The fourth, N, is a numeric variable, and the fifth, A( ), is a numeric array. Empty parentheses are used to indicate that the entire array is to be saved. Each individual item in the DATASAVE argument list (including each array element) is considered to be a single argument. Thus, if the array A( ) is dimensioned to contain four elements, the DATASAVE DC argument list in statement 20 consists of a total of eight data values.

When the DATASAVE DC statement is executed, the arguments are taken in sequence from the argument list and stored in a logical record on the disk (if a two-dimensional array is included in the argument list, the array elements are transferred row by row). Thus, if the following assignments are assumed, the logical record created by statement 20 resembles the figure below.

```
B$    =  "10 OAK DRIVE"
N     =  2222
A(1)  =  123
A(2)  =  456
A(3)  =  789
A(4)  =  100
```

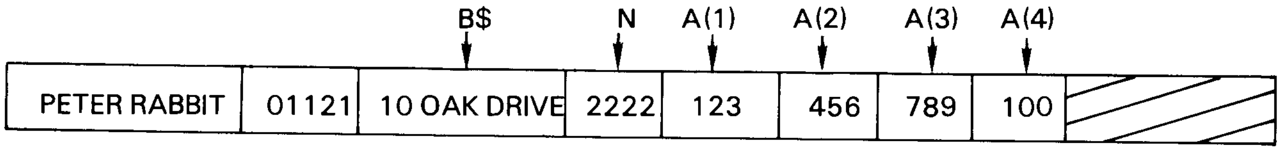


Figure 2-5. Logical Record Consisting of One Sector

The arguments saved in a logical record on disk are commonly referred to as fields within the record. In the previous record, for example, "PETER RABBIT" is the first field in the record, while '100' is the last field. It is important to note that when a logical record is read back into memory from disk, each field must be read into a single variable or array element; it is never possible to read two or more fields into a single variable or array element, even if the receiving variable or element is large enough to contain more than one field. (The PACK and $PACK statements can be used to store several values in a single alpha variable which can be saved in a single field within a record. UNPACK and $UNPACK convert a packed value into several values for a list of variables.) Note, too, that alphanumeric fields must be read back into alphanumeric variables or array elements, while numeric fields must be read back into numeric variables or array elements.

In the present example, the logical record occupies somewhat less than one sector. Notice in Figure 2-5 that the remainder of the sector is unused. The remainder of the sector contains meaningless data, which is ignored by the system (the system provides automatic safeguards against accidentally reading this meaningless data when the catalog procedures are used). If another logical record is created (with a second DATASAVE DC statement), the new record begins at the beginning of the next sector. The remaining unused portion of the first sector is not used for the second record. A logical record always begins at the beginning of a sector. This is the case even if the logical record occupies only a very small portion of a sector. For example, consider the statements:

```
10 A = 34.2
20 B = 100
30 DATASAVE DC A
40 DATASAVE DC B
```

Each of these statements creates a single logical record containing a single numeric value, and each occupies an entire sector on the disk:

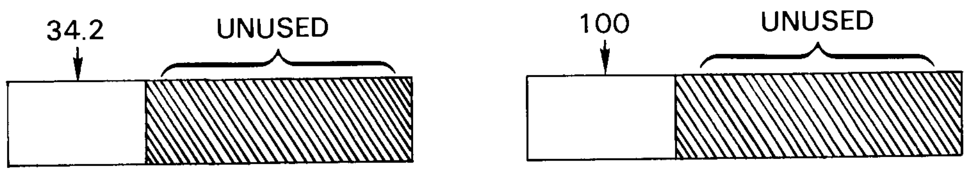


Figure 2-6. Two One-Sector Logical Records

Obviously, this is not a very efficient way to store data. It would surely be more efficient to store both values in a single record, using a single DATASAVE DC statement:

```
20 DATASAVE DC A, B
```

On the opposite end of the spectrum, a single logical record can occupy several sectors - as many sectors, in fact, as are required to store all the data in the DATASAVE DC argument list. The programmer need not be concerned about multiple sector records, since the system automatically appropriates as many sectors as it needs to store a logical record.

Payroll files and inventory files are two examples of files which commonly contain multiple-sector records. In such files, each record may be several hundred bytes in overall length, and may occupy three or more sequential sectors. For example, a comprehensive inventory record might be made up of three large arrays, A$( ), B$( ), and C$( ), each 200 bytes in length:

```
10 DIM A$(4)50, B$(2,2)50, C$(2)100
:
200 DATASAVE DC A$( ), B$( ), C$( )
```

The logical record created on disk by the DATASAVE DC statement at line 200 consists of three consecutive sectors, and looks like this:

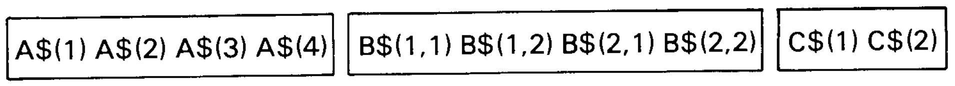


Figure 2-7. Logical Record Consisting of Three Sectors.

Notice particularly that in sector #2 of this record, the two-dimensional array B$( ) is stored row by row.

In addition to the user's data stored in each sector of a record, the system itself records several bytes of control information. These control bytes are completely transparent to the user, but they occupy storage bytes in the sector, and must be included when figuring the total number of sectors required for a record of determinate length. The space allocation of data in sectors of a logical record and the control bytes are explained in Chapter 4.

2.14 OPENING A SECOND DATA FILE ON DISK

Most applications require the maintenance of two or more data files on disk. An update operation may require three files (old master, new master, and transaction file). The process of opening a second, and subsequent, data file on disk is identical to that of opening the first. The DATASAVE DC OPEN statement is used. For example, the following statement could be used to open a second data file, named "DATFIL-2" on the 'F' platter:

```
250 DATASAVE DC OPEN F (500) "DATFIL-2"
```

However, it is important to note that in opening DATFIL-2, the system automatically closes DATFIL-1, and DATFIL-2 now becomes the currently open file on disk. Any DATASAVE DC or DATALOAD DC statement therefore automatically accesses DATFIL-2 instead of DATFIL-1. Chapter 3 introduces a technique for keeping more than one file open on disk at the same time. For the present, however, it is assumed that only one file can be the currently open file at any given moment.

2.15 RE-OPENING A DATA FILE ON DISK: THE "DATALOAD DC OPEN" STATEMENT

After a data file has been created on disk with DATASAVE DC OPEN and subsequently "closed" by opening a second file, the data in the original file can be accessed by re-opening the file with a DATALOAD DC OPEN statement. DATALOAD DC OPEN is used to re-open an existing file irrespective of whether existing data is to be read from the file, or additional data is to be stored in it. The DATASAVE DC OPEN statement used to open a new file initially, cannot be used to re-open an existing file; any attempt to use this statement to re-open a file will result in an error.

In the DATALOAD DC OPEN statement, the system must be provided with the following information:

1. The disk platter on which the file is cataloged.
2. The name of the file. The name may be specified as a literal string ("DATFIL-1") or as the value of an alpha variable (A$ = "DATFIL-1").

When a DATALOAD DC OPEN statement is executed, the system searches the Catalog Index on the designated disk platter for the specified file name. The file's location is then recorded in memory for future reference to the file.

Example 2-26: Re-Opening a Cataloged Data File with DATALOAD DC OPEN

```
300 DATALOAD DC OPEN F "DATFIL-1"
```

Statement 300 causes the system to search the Catalog Index on the 'F' platter for the file name "DATFIL-1". The file's location is then stored in memory for future reference.

Example 2-27: Re-Opening a Cataloged Data File on the Model 2280 with DATALOAD DC OPEN

```
100 SELECT #4/D11
200 DATALOAD DC OPEN T #4, "PROB-2"
```

Statement 100 assigns the fixed platter address D11 to the #4 slot in the Device Table (see Section 3.2). Statement 200 causes the system to search the Catalog Index on the D11 platter for the file name "PROB-2". The file's location is then stored in memory for future reference.

Of course, the file name specified in the DATALOAD DC OPEN statement must be the name of a data file currently cataloged on the specified disk platter. If the system cannot locate the file name in the Catalog Index, an error is signalled.

Example 2-28: Attempting to Re-Open a Non-Cataloged Data File

```
300 DATALOAD DC OPEN F "DOTFIL-1"

                                   ↑ERR D82
```

Statement 300 attempts to re-open a data file whose name is not listed in the Catalog Index of the platter. Since "DOTFIL-1" is not identical to "DATFIL-1" Error D82 (File Not In Catalog) is signalled.

Once a file has been re-opened with a DATALOAD DC OPEN statement, it is possible both to store new data in the file (with a DATASAVE DC statement), and to read existing data from the file (with a DATALOAD DC statement); see Section 2.16.

2.16 RETRIEVING DATA FROM A CATALOGED DATA FILE ON DISK: THE "DATALOAD DC" STATEMENT

Data stored on a disk has little value if it cannot be located and read back into memory for analysis and processing. In Catalog mode, data is read from a currently open file on disk with a DATALOAD DC statement. When loading data from the disk into memory, you must tell the system which variable(s) and/or array(s) in memory are to receive the data. The list of receiving variables and arrays is specified in a DATALOAD DC statement, and is known as the argument list for that statement. The system reads one or more logical records from the currently open file on disk (if no file is currently open, an error is signalled), and sequentially stores the data in the variable(s) and array(s) specified in the argument list. The system continues to read data from the file until all variables in the argument list have been filled, or until there is no more data remaining in the file. If the argument list contains more receiving variables than there are fields in a record, the first fields of the next sequential record are automatically read to satisfy all unfilled variables. The remainder of the second record is then read and ignored. If only the first few fields in a record are read (i.e., if the argument list contains fewer receiving arguments than there are fields in the record), the remainder of the record is read but ignored.

Example 2-29: Reading Data from a Cataloged Data File with DATALOAD DC

```
310 DIM B(60)
320 DATALOAD DC B()
```

At line 310 a numeric array B( ) is dimensioned to hold 60 elements. At line 320, the system is instructed to load 60 numeric values from the currently open file on disk into array B( ).

It is important to recognize that the quantity of data read by a DATALOAD DC statement is determined solely by the number of receiving arguments in the DATALOAD DC argument list, and not by the length of a logical record in the file. DATALOAD DC reads exactly enough data to satisfy its argument list, whether that means reading only a portion of one record, more than one record, or exactly one record.

If the DATALOAD DC argument list requires less than an entire logical record, the remaining unused data in the record is read but ignored, and the system ends up positioned at the beginning of the next logical record. This is true even for multi-sector records; if only the first sector of a multi-sector record is read, the remaining sectors are skipped, and cannot be retrieved unless the entire record is reread. The next DATALOAD DC statement automatically begins reading with the next sequential logical record.

Example 2-30: Reading Less Than One Logical Data Record with DATALOAD DC

```
10   DIM A$(4)50, B$(2,2)50, C$(100)
:
40   DATASAVE DC OPEN F (100) "DATFIL-1"
50   DATASAVE DC A$( ), B$( ), C$( )
:
:
200  DATALOAD DC OPEN F "DATFIL-1"
210  DATALOAD DC A$( )
```

In this example, a record consisting of three sectors (600 bytes) is written at line 50. At line 210, a DATALOAD DC statement reads only 200 bytes of data (the first sectors) from this record. The remaining two sectors of the record are ignored, and the system is positioned at the beginning of the next logical record in DATFIL-1.

If, on the other hand, the DATALOAD DC argument list requests more than an entire logical record, data is appropriated from the next sequential logical record to satisfy the argument list. In this case as in the preceding case, if only a portion of the second record is read, the remaining unread portion is ignored, and the system is positioned at the beginning of the third logical record.

Example 2-31: Reading More Than One Logical Data Record with DATALOAD DC

```
10   DIM A$(5), B$(5)
:
40   DATASAVE DC OPEN F (100) "DATFIL-1"
50   DATASAVE DC A$( )
60   DATASAVE DC B$( )
:
:
200  DATALOAD DC OPEN F "DATFIL-1"
210  DATALOAD DC A$( ), B$( )
```

A pair of logical records, the first containing A$( ) and the second B$( ), are written into DATFIL-1 at lines 40 and 50. At line 210, both logical records are read back into memory by the same DATALOAD DC statement.

While the versatility of the DATALOAD DC statement may be important for specialized access routines, it is generally good programming procedure to read back exactly one logical record with each DATALOAD DC statement. To accomplish this, the DATALOAD DC argument list used to read a record must correspond to the DATASAVE DC argument list used in writing the record. It is not required that the two argument lists be identical, however; the only requirement is that the same number of fields be read as were written in the record initially. For example, the record created with the statement

```
50 DATASAVE DC A$, B$, C$, D$
```

might be read with the statements

```
200 DIM F$(4)
210 DATALOAD DC F$()
```

In this case, the original content of A$ is returned to F$(1), of B$ to F$(2), of C$ to F$(3), and of D$ to F$(4).

Similarly, the record created by the statements

```
10 DIM A$(4)
:
:
50 DATASAVE DC A$()
```

could be read with the statement

```
200 DATALOAD DC B$, C$, D$, E$
```

There are however, several rules which govern the lengths and types of receiving arguments in the DATALOAD DC argument list:

1. Only one field (value) in a record can be returned to each receiving argument in the DATALOAD DC argument list. This rule holds true even if the receiving argument is dimensioned large enough to hold two or more fields. For example, assume a record containing two fields is stored on disk:

```
10 DIM A$3, B$3
20 A$ = "ABC" :B$ = "EFG"
30 DATASAVE DC A$, B$
```

Subsequently, an attempt might be made to read this record with the statements

```
200 DIM C$6
210 DATALOAD DC C$
```

The result of this operation, however, is that C$ = "ABC ". The second field, EFG, is not read into C$, despite the fact that C$ is large enough (6 bytes) to store both ABC and EFG.

2. If, as in the preceding case, a character string is shorter than the length of the receiving alphanumeric argument in which it is stored, the remaining unfilled bytes of the argument are initialized to contain blanks. For example, a single field consists of the three characters ABC, and the following statements are executed to read it:

```
200 DIM C$6
210 DATALOAD DC C$
```

The result is that the first three bytes of C$ contain the letters ABC, and the last three bytes contain blanks.

3. If, on the other hand, a character string is longer than the length of the alpha argument in which it is stored, the value is truncated, and the excess characters are lost. For example, a single field consists of the six characters ABCDEF, and the following statements are executed to read it:

```
200 DIM C$3
210 DATALOAD DC C$
```

The result is that C$ contains the characters ABC. The remaining three characters in the string are lost. (They remain, of course, in the record on disk.)

4. Numeric values must be returned to numeric receiving arguments. Alphanumeric values must be returned to alpha arguments; attempts to store an alpha value in a numeric argument generate an Error X74 (Wrong Variable Type) and terminate program execution. For example, a record might be created consisting of one numeric and one alphanumeric field:

```
10 A$ = "ABC"
20 N = 1234
30 DATASAVE DC A$, N
```

This record could be retrieved with a statement of the form

```
200 DATALOAD DC A$, X
```

However, a statement of the form

```
200 DATALOAD DC M,N
```

will generate an Error X74, since the alphanumeric value ABC cannot be returned to a numeric variable (M).

2.17 SKIPPING AND BACKSPACING OVER LOGICAL RECORDS IN A CATALOGED DATA FILE: THE "DSKIP" AND "DBACKSPACE" STATEMENTS

An existing data file on the disk is generally re-opened (with a DATALOAD DC OPEN statement) for one of three reasons:

1. To read data from the file.
2. To store additional data in the file.
3. To change or update existing data in the file.

In any of these three cases, it is usually necessary to access one or more specific logical records within the file. Two catalog statements, DSKIP and DBACKSPACE, enable you to move through a file and access particular records within the file.

The use of DSKIP and DBACKSPACE can be illustrated by considering a file which consists of several logical records:

```
400 DATASAVE DC OPEN F (50) "TEST-1"
410 DATASAVE DC A( )
420 DATASAVE DC B( )
430 DATASAVE DC C( )
440 DATASAVE DC D( )
450 DATASAVE DC E( )
460 DATASAVE DC END
```

This file, named "TEST-1" occupies 50 sectors on the 'F' platter. Five logical records (statements 410-450) have been stored in TEST-1, and a trailer record has been written following the last logical record. Assuming that each logical record consists of two sectors, the five records occupy ten sectors (see Figure 2-8).

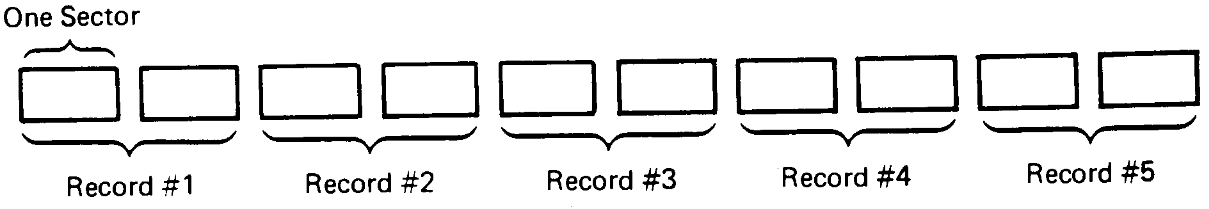


Figure 2-8. Logical Records in TEST-1

Suppose, now, that TEST-1 is closed and subsequently re-opened with a DATALOAD DC OPEN statement. When the file is re-opened, the system automatically positions itself at the beginning of the file. In order to access any record other than record #1, you must instruct the system to skip ahead through the file to the desired record. You can skip over logical records in a data file with a DSKIP statement. In the DSKIP statement, you must tell the system how many records to skip. Suppose, for example, you wish to read record #3 in the file. Since the system is currently positioned at record #1, it is necessary to skip two records. (See Figure 2-9.)

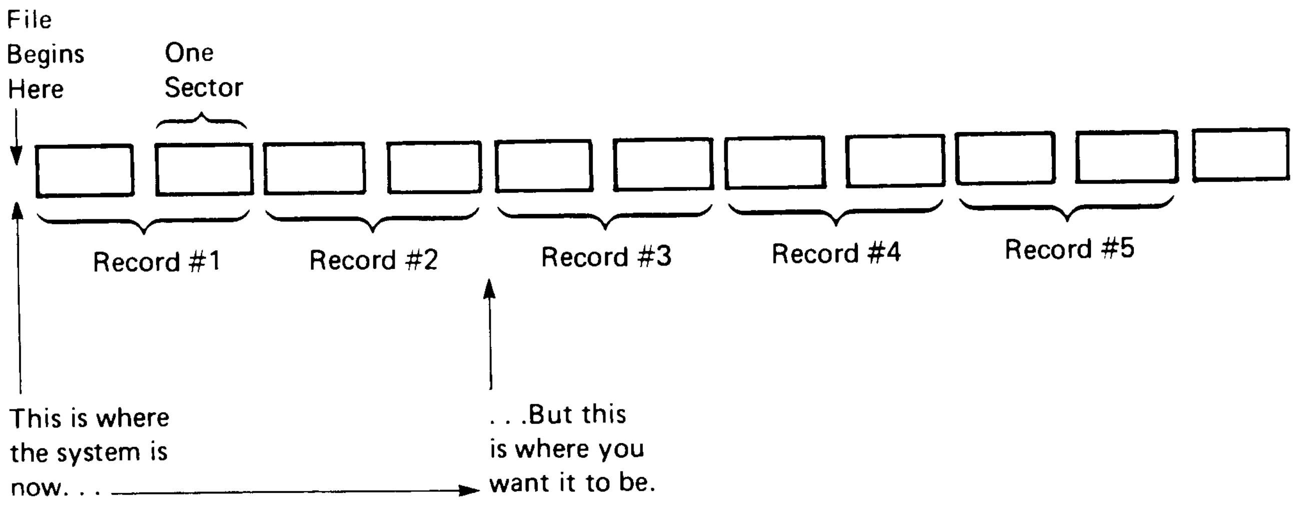

File Begins Here
One Sector
Record #1
Record #2
Record #3
Record #4
Record #5
This is where the system is now. . .
. . .But this is where you want it to be.

Figure 2-9. Skipping over Logical Records in a Data File

Example 2-32: Skipping over Logical Records in a Data File on a Model 2280 Platter

```
460 SELECT #2/D12
470 DATALOAD DC OPEN T #2, "TEST-1"
480 DSKIP 2
```

Statement 470 re-opens TEST-1 on the D12 platter. The system is positioned at the beginning of the file. Statement 480 instructs the system to skip two logical records (records #1 and #2), and reposition itself at the beginning of record #3. A DATALOAD DC statement such as

```
490 DATALOAD DC C()
```

now loads record #3 from the file into memory.

Notice that the number supplied in the DSKIP statement specifies how many logical records are to be skipped (remember that each logical record was created by a single DATASAVE DC statement). It does not matter how many sectors are contained in each logical record (record #1 might contain five sectors, for example, while record #2 contains ten, etc.). Be sure, however, that the argument list of the DATALOAD DC statement which is used to load a record is identical to the argument list of the DATASAVE DC statement which originally created the record.

After a logical record has been loaded, the system is positioned at the beginning of the next logical record. Suppose that you now want to load and check logical record #1 from TEST 1. Since the system is currently positioned at the beginning of record #4 (having just loaded record #3), you must backspace three logical records (see Figure 2-10). You can do so with a DBACKSPACE statement.

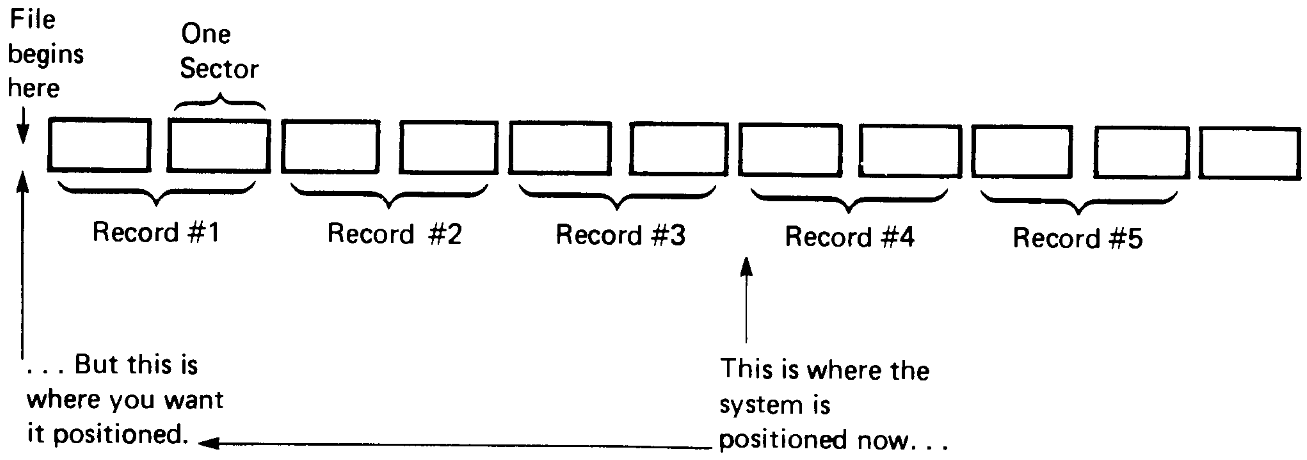

File begins here
One Sector
Record #1
Record #2
Record #3
Record #4
Record #5
. . . But this is where you want it positioned.
This is where the system is positioned now. . .

Figure 2-10. Backspacing over Logical Records in a Data File

Example 2-33: Backspacing over Logical Records in a Data File

```
500 DBACKSPACE 3
```

Statement 500 causes the system to backspace over three logical records in the currently open file (TEST-1) on disk. Since the system is currently positioned at the beginning of record #4, it is repositioned to the beginning of record #1 following statement execution. Record #1 can now be loaded with a DATALOAD DC statement such as

```
510 DATALOAD DC A()
```

It is possible to backspace to the beginning of a file from any point in the file with a DBACKSPACE BEG statement. In Example 2-33, for example, it would have been just as easy to access record #1 by backspacing to the beginning of the file and executing statement 510.

Example 2-34: Backspacing to the Beginning of a Cataloged Data File

```
500 DBACKSPACE BEG
```

Statement 500 instructs the system to backspace from its current position in the file to the beginning of the file (i.e., the beginning of the first record of the file).

In order to store additional data in a file which has just been re-opened, it is necessary to skip to the current end of the file (that is, the end of all currently-stored data in the file) and begin saving the new data at that point. This can be done with a DSKIP END statement, if the current end of file is marked by an end-of-file trailer record (written with DATASAVE DC END). If no end-of-file trailer record has been written in the file, however, an ERROR D87 (No End of File) is returned following execution of the DSKIP END statement. The DSKIP END statement locates the end-of-file trailer record, and repositions the system at the beginning of the trailer record in the file. A new data record can then be saved over the trailer record, and a new trailer record written to mark the new end of the file.

Example 2-35: Skipping to the End of a Cataloged Data File

```
520 DSKIP END
```

Statement 520 instructs the system to skip to the current end of the currently open data file on the disk (TEST 1). A trailer record must have been written in the file with a DATASAVE DC END statement (statement 460) following the most recent DATASAVE statement (statement 450); otherwise, an ERROR D87 is returned. After the DSKIP END statement is executed, the system is positioned at the beginning of the trailer record in the file. A new data record can be saved over the trailer record, and a new trailer record written in the file, with the following statements:

```
530 DATASAVE DC F()
540 DATASAVE DC END
```

2.18 TESTING FOR THE END-OF-FILE

The presence of an end-of-file (EOF) trailer record in a file makes it possible to test for an end-of-file condition when reading the file. The end-of-file test is made with the IF END THEN statement, which initiates a program branch to a specified line number when the EOF record is read. This technique enables the programmer to construct a loop which will read and process data from a file sequentially, independent of the number of records in the file. When the EOF record is read, the read-and-process routine is terminated, and program execution resumes at another point in the program.

Example 2-36: Testing for the End-Of-File Condition in a Cataloged Data File

```
600 DATALOAD DC OPEN F "TEST-1"
610 DATALOAD DC A$(), B$(), C$()
620 IF END THEN 700
:
     (process data from A$(), B$(), C$())
:
660 GOTO 610
700 STOP
```

Statement 600 opens the file TEST-1, and statement 610 loads a logical record from that file into the arrays A$( ), B$( ), and C$( ). Statement 620 then tests for the data trailer record signifying that the last data record in the file has been read. If this is the case, the program jumps to statement 700 and stops. If it is not the case, the data loaded into the three arrays is processed until, at statement 660, the system is instructed to loop back and load in another record.

Three important points must be noted regarding the EOF testing procedure:

1. The file being read must have an EOF record as the trailer record. (Produced by the DATASAVE DC END statement.)

2. The EOF record is not transferred into the DATALOAD DC argument list when it is read with a DATALOAD DC statement. Detection of the EOF record invokes a special routine which stores the EOF data in a special section of memory reserved for system use. The contents of the DATALOAD DC argument list are therefore unchanged following retrieval of the EOF, and still contain the values read from the last data record. The EOF record itself is inaccessible to the programmer in catalog mode.

3. Following a read of the EOF record, the system automatically moves itself back one sector, and ends up positioned at the beginning of the EOF record rather than at the beginning of the next sequential sector. This feature enables the programmer to immediately begin saving additional records in the file by writing over the EOF record. A new EOF record should, of course, be written following the last new data record.

2.19 SCRATCHING UNWANTED FILES

A data file or program file which has outlived its usefulness presents certain problems to the programmer. In the first place, such files hold the potential for disaster if the operator should inadvertently read and process data from an obsolete data file, or load and run an obsolete program. In the second place, obsolete files occupy valuable disk space which the programmer will surely want to reuse. The SCRATCH statement provides solutions to both of these problems.

The SCRATCH statement sets the status of the named file to a scratched condition. A scratched file is not physically removed from the disk. The file's name and location remain listed in the Catalog Index, but the file is flagged as a scratched file. A scratched file has two significant characteristics:

1. A scratched data file cannot be re-opened with DATALOAD DC OPEN, and a scratched program file cannot be loaded into memory with LOAD or LOAD RUN. The danger of accidentally accessing a scratched file is therefore removed.

2. A scratched file can, however, be renamed and re-opened with a DATASAVE DC OPEN or SAVE statement. In this case, a new file is created in the space previously occupied by the scratched file. The procedure for renaming a scratched file is covered in Chapter 4.

3. When a disk platter is backed up by copying it to another platter with a MOVE command, all scratched files are automatically removed. This provides the overall capability of freeing all scratched disk space in a contiguous manner.

Example 2-37: Scratching Unwanted Files

```
750 SCRATCH F "PROG1" "TEST1"
```

Statement 750 sets the status of the program file PROG1 and the data file TEST1 to a scratched condition. PROG1 cannot be loaded into memory with a LOAD statement, and TEST1 cannot be opened to load or save data with a DATALOAD DC OPEN statement. New files can be stored in the space occupied by PROG1 and TEST1, however. (Refer to Chapter 4 for a discussion of how to re-use the space occupied by scratched files.)

If a LIST DC F statement is executed following statement 750 in Example 2-37 above, the Catalog Index listing looks like this:

```
INDEX SECTORS = 00100
END CAT. AREA  = 01000
CURRENT  END  = 00269

                     NAME      TYPE   START    END      USED     FREE
These files          DATFIL1    D     00100    00199    00002    00098
are  ──────────────► TEST1      SD    00200    00249    00001    00049
scratched ─────────► PROG1      SP    00250    00269    00020    00000
```

Figure 2-11. The Catalog Index Showing Scratched Files

Notice that under "TYPE" PROG1 reads "SP" and TEST1 reads "SD". The "S" signifies that each file has been scratched.

The disk space occupied by scratched files can be reused in two ways. Firstly, the MOVE statement can be used to copy the entire catalog onto a second platter, automatically deleting all scratched files in the process. MOVE is discussed in the following section. Secondly, individual scratched files can be renamed and their locations reused by new files with a DATASAVE DC OPEN or SAVE statement. The process of renaming and reusing scratched file space is covered in Chapter 4.

2.20 MAKING BACKUP COPIES OF CATALOGED FILES: THE "MOVE" STATEMENT

MOVE and COPY operations allow the transfer of information (platter-to-platter) between separate platters of a single disk unit or between separate disk units. In general, the discussion of disk statements and commands which follows applies across the board to all disk models (except 2270A when using IBM type diskettes), since all 2200 series disks share the same BASIC instruction set.

Backing Up Disk Files

It is important that backup copies of all valuable disk files be regularly maintained. Disk platters, like other storage media, are subject to wear given excessive use, and there is always the possibility of accidental damage or destruction. The effects of a data loss due to accident or wear can be disastrous unless backup platters have been maintained; the cost of buying, storing, and regularly updating backup platters is therefore more than offset by their potential value in the event of such a catastrophe.

The frequency with which backup copies of an individual file must be created differs from case to case and is determined by the degree of activity of the file and the method of updating it. In a typical commercial application, transactions are accumulated over a definite period in a separate transaction file, and posted as a batch to the master file at the end of the period. A completely new master file is created (usually on a separate platter) as a result of the batch posting operation. In this case, it is a good rule of thumb to retain the most recent "generation" of the master file (the original master file) as a backup for the new master file. The transaction file may or may not be retained, depending upon the degree of difficulty involved in reproducing it.

When the update process involves recreation of the master file on a separate platter, the old master file serves as an automatic backup. In many cases, however, a file's degree of activity does not warrant a batch processing approach. Relatively inactive files (inventory files are a common example) often require so few updates during the course of a monthly or quarterly period that it is simply inefficient to reproduce the entire file in the process of posting a handful of changes. Updates to such files are frequently done "in place" in the original file, with the result that a separate copy of the master file is not produced. A copy operation is required in this situation to backup the master file on a second disk platter. The MOVE statement can be used for this purpose.

The MOVE statement has the capability to copy an individual cataloged file onto a separate platter (if the file name is specified), or to copy the entire contents of the catalog onto a second platter (if no individual file name is specified). In the latter case, MOVE performs the additional function of deleting all scratched files from the catalog when it is copied to the new platter.

MOVE effectively has two separate forms:

Form #1 -	if the MOVE statement includes no reference to a specific file name, the entire catalog (including the Catalog Index and Catalog Area) is copied from the origin platter to the destination platter, and scratched files are deleted.
Form #2 -	if a specific file name is referenced in the MOVE statement, that file is copied from the catalog on the origin platter to the catalog on the destination platter.

Moving the Entire Catalog (Form #1 of MOVE)

In Form #1 of the MOVE statement, no file name is specified. The absence of an individual file name indicates that the entire catalog is to be moved. This form of MOVE performs the following functions:

1. Creates a backup copy of the entire catalog on a second platter

2. Eliminates all scratched files from the backup catalog, and rearranges still-active files to fill the available space, thus making more efficient use of the Catalog Area and Catalog Index on the second (destination) platter.

The MOVE statement copies the entire catalog from the origin platter to the destination platter, removing all scratched files from the Catalog Area and deleting scratched file names from the Catalog Index in the process. After all scratched files are removed, all remaining active files are moved up to fill the vacated sectors in the Catalog Area, and the Catalog Index is revised to reflect the files' new positions in the Catalog Area.

The only requirement of the second (destination) platter is that it be formatted. The destination platter may be in the same disk unit as the origin platter, or in a separate disk unit on the same system. The user is not required to set up a catalog on the destination platter with SCRATCH DISK, since the MOVE statement automatically establishes a Catalog Index and Catalog Area on the destination platter identical in size to the originals, before it begins copying files. The sizes of the Catalog Index and Catalog Area are therefore not altered by MOVE.

Note that Form #1 of the MOVE statement is non-destructive for the origin platter, but destructive for the destination platter. The original catalog is not altered by the MOVE operation; all of its files (including all scratched files) remain exactly as they were prior to the MOVE. Information already recorded on the destination platter is destroyed when the catalog is copied on that platter, however, unless it is outside the range of the Catalog Area. For example, if the END sector of the Catalog Area is 1000, sectors 0-1000 on the destination platter are overwritten and destroyed when the catalog is moved. Sectors from 1001 onward on the destination platter are unaffected by the MOVE, however.

Example 2-38: Moving the Catalog from One Platter to Another, and Deleting Scratched Files with MOVE

```
450 MOVE F TO R
```

Line 450 copies the contents of the catalog from the 'F' platter to the 'R' platter, automatically squeezing out all scratched files (see Figure 2-12).

Example 2-39: Moving the Catalog and Deleting Scratched Files with MOVE Statement on the Model 2280

```
350 MOVE T/D12, TO T/D10,
```

Line 350 copies the contents of the catalog from the fixed platter designated D12 to the removable platter (D10), automatically squeezing out all scratched files.

Original Catalog on F Platter

```
INDEX SECTORS = 00024
END CAT. AREA  = 01023
CURRENT END    = 00089
```

NAME	TYPE	START	END	USED	FREE
DATA-L1	P	00067	00070	00004	00000
2231W	P	00077	00079	00003	00000
XPRINT	P	00080	00082	00003	00000
JUNK	D	00030	00039	00009	00001
FLIPS	SP	00027	00029	00003	00000
PASCAL	P	00024	00026	00003	00000
INVTORY	D	00046	00055	00005	00005
CREATE	P	00056	00059	00004	00000
INVTY-P	P	00071	00073	00003	00000
FLIPS-1	P	00083	00086	00004	00000
CREATE-1	P	00060	00063	00004	00000
DATA-S1	P	00064	00066	00003	00000
DATA-S	SP	00040	00042	00003	00000
2221W-B	P	00074	00076	00003	00000
DATA-L	SP	00043	00045	00003	00000
ABCDEF	P	00087	00089	00003	00000

New Catalog on R Platter

```
INDEX SECTORS = 00024
END CAT. AREA  = 01023
CURRENT END    = 00089
```

NAME	TYPE	START	END	USED	FREE
DATA-L1	P	00024	00027	00004	00000
2231W	P	00028	00030	00003	00000
XPRINT	P	00031	00033	00003	00000
JUNK	D	00034	00043	00009	00001
PASCAL	P	00044	00046	00003	00000
INVTORY	D	00047	00056	00005	00005
CREATE	P	00057	00060	00004	00000
INVTY-P	P	00061	00063	00003	00000
FLIPS-1	P	00064	00067	00004	00000
CREATE-1	P	00068	00071	00004	00000
DATA-S1	P	00072	00074	00003	00000
2221W-B	P	00075	00077	00003	00000
ABCDEF	P	00078	00080	00003	00000

Notes:

1) 1000 sectors were originally reserved in the catalog area with SCRATCH DISK F END = 1023.

2) Programs 'FLIPS' 'DATA-S' and 'DATA-L' were scratched with SCRATCH F "FLIPS" "DATA-S" "DATA-L".

Figure 2-12. Original Catalog and Copied Catalog Following a MOVE, Showing Scratched Files Deleted

NOTE TO MODEL 2270-3 AND 2270A-3 USERS

Moving a catalog to or from the third drive of a triple removable diskette drive calls for use of the 'F' parameter and the disk device address 350. The following statement moves a catalog from the first platter to the third in a triple drive unit:

```
500 MOVE F/310, TO F/350,
```

For Model 2260 and 2270 Series operations, if the origin and destination platters are located in separate disk units, the appropriate disk device addresses must also be specified.

Example 2-40: Moving the Catalog from One Disk Unit to Another with MOVE

```
550 MOVE R /310, TO R /320,
```

Line 550 causes the catalog to be moved from the 'R' platter in the disk unit with address 310 to the 'R' platter in the disk unit on the same system with address 320. Scratched files are, of course, deleted.

Because address 310, the default address, is assumed if no address is specified, it need not be referenced explicitly. For example, line 550 could be rewritten as follows:

```
550 MOVE R TO R /320,
```

In this case, the address syntax /310 preceding "TO" is assumed, and the statement functions exactly as described in Example 2-40.

Moving Individual Cataloged Files (Form #2 of MOVE)

If an individual file name is specified in a MOVE statement, the named file is copied from the origin platter to the destination platter. In this case, MOVE does not initialize a catalog on the destination platter; the catalog must be initialized by the user (with SCRATCH DISK) prior to attempting the MOVE. MOVE simply copies the named file from the Catalog Area on the original catalog to the Catalog Area on the destination platter, and enters the file name in the Catalog Index on the destination platter. If the destination platter does not have a catalog or if the catalog is not large enough to accommodate the file to be moved, an error is signalled, and the MOVE is aborted.

Example 2-41: Moving a Named Cataloged File from One Platter to Another with MOVE

```
250 MOVE F "PROG#1" TO R
```

Line 250 copies the cataloged file PROG#1 from the catalog on the 'F' platter to the catalog on the 'R' platter. The file is recorded in the next sequential available location in the Catalog Area on the 'R' platter, and its name ("PROG#1") is inserted in the 'R' platter's Catalog Index.

Example 2-42: Moving a Named Cataloged File from One Model 2280 Platter to Another with MOVE

```
200 MOVE T/D15, "PROB#8" TO T/D13,
```

Line 200 copies the cataloged file "PROB#8" from the catalog of the D15 platter to the catalog of the D13 platter. The file is recorded in the next sequential available location in the Catalog Area on the D13 platter, and its name ("PROB#8") is inserted in the D13 platter's Catalog Index.

As with Form #1 of MOVE, Form #2 can be used to copy a file from platter to platter within the same disk unit, or from one disk unit to another on the same system.

Example 2-43: Moving a Named Catalog File from One Disk Unit to Another with MOVE (Model 2260 and 2270 Series)

```
270 MOVE R /320, "PROG#2" TO F /310,
```

Line 270 copies the cataloged file PROG#2 from the 'R' platter of the disk unit with address 320 to the 'F' platter of the disk unit with address 310.

If Form #2 of MOVE is used to copy a single file, the user also has the option to expand the size of the file when it is copied to the destination platter. A numeric expression, enclosed in parentheses and immediately following the destination platter parameter in a MOVE statement, informs the system that additional sectors are to be reserved for the file when it is copied in the Catalog Area on the destination platter. The truncated value of the expression indicates the number of additional sectors to be reserved.

Example 2-44: Copying a Single Named Cataloged File from One Platter to Another with MOVE (Additional Sectors Reserved)

```
390 MOVE R "DATFIL1" TO F (10)
```

Line 390 causes the file DATFIL1 to be copied from the catalog on the 'R' platter to the catalog on the 'F' platter. The numeric value '10' in parentheses following the destination platter 'F' indicates that 10 additional sectors are reserved for DATFIL1 when it is copied on the "F" platter.

NOTE:

A file may never be compressed when it is moved to the destination platter. Negative values of the numeric expression are illegal.

2.21 TESTING THE VALIDITY OF DISK SECTORS: THE "VERIFY" STATEMENT

Its reliability as an external storage device is an important feature of the disk drive, and this reliability is augmented by the automatic redundancy checks performed by the system, as well as by the optional read-after-write check specifiable by the programmer. Despite these safeguards, however, there are cases where it is desirable to be able to test particular sectors directly, with a special statement. The VERIFY statement provides such a capability.

VERIFY performs a Cyclic Redundancy Check (CRC) and Longitudinal Redundancy Check (LRC) on each specified sector. A CRC checks the recording of data in each sector while the LRC checks the transfer of data between the CPU and the disk processing unit. The checks performed by VERIFY are particularly useful immediately following a MOVE, when they can be employed to verify the accuracy of the data copied to the destination platter. VERIFY also is a useful diagnostic tool for testing a platter which has been causing read/write problems, or for checking the data already recorded on a platter prior to accessing it. Many programmers verify important platters regularly at the beginning of daily operation. The CRC and LRC checks performed by VERIFY provide a means of detecting recording errors immediately after recording rather than at some later time.

VERIFY, like MOVE, is a single statement with two alternative forms:

Form #1 - No receiving variable specified in the VERIFY statement. In this case, VERIFY checks the entire range of specified sectors, displaying error messages for all sectors which do not verify.

Form #2 - Receiving variable included in VERIFY statement. In this case VERIFY checks the specified range of sectors until it encounters an invalid sector. At that point, the VERIFY operation terminates, and the receiving variable is set to the address of the next sequential sector following the invalid sector. No error message is displayed. If all sectors verify correctly, the receiving variable is set to zero.

Form #1 of VERIFY (No Receiving Variable Specified)

If the VERIFY statement does not include a receiving numeric variable, Form #1 is assumed. Form #1 causes the system to verify a specified range of sectors on a designated platter, automatically displaying error messages for any sectors which do not verify.

Error messages are of the form:

```
ERROR IN SECTOR 995
```

The programmer may specify the range of sectors to be verified by including the starting and ending sector addresses in the VERIFY statement. All sectors between and including the specified sectors are verified.

Example 2-45: Verifying a Specified Range of Sectors with VERIFY (Form #1)

```
300 VERIFY R (0, 1000)
```

Line 300 instructs the system to verify sectors #0 through #1000 on the 'R' platter. The entire 1001 sectors are verified, and an error message is displayed or printed for each sector which does not verify.

VERIFY is typically used to check the catalog on the destination platter following a MOVE or COPY (see Section 6.6). In this case, the starting and ending sector addresses are omitted; the system automatically verifies up to the current catalog end, beginning with section #1 (first sector of the catalog index), and ending with the current catalog end. Note that if the sector address parameters are omitted and VERIFY is instructed to check a platter which does not contain a catalog, the system either displays an error message and aborts the VERIFY operation, or it may verify arbitrary sectors.

Example 2-46: Verifying the Catalog with VERIFY (Form #1) (Model 2260 and 2270 Series)

```
250 MOVE F TO R
260 VERIFY R
```

Line 250 copies the catalog from the 'F' platter to the 'R' platter. Line 260 verifies the new catalog on the 'R' platter. Because no sector addresses are included in the VERIFY statement, all sectors up to and including the current catalog end are automatically verified. Each invalid sector produces an error message on the Console Output device.

Example 2-47: Verifying the Catalog with VERIFY (Form #1) (Model 2280 Series)

```
150 MOVE T/D14, TO T/D10,
200 VERIFY T/D10,
```

Line 150 copies the catalog from the D14 fixed platter to the D10 removable platter. Line 200 verifies the new catalog on the D10 platter.

Form #2 of VERIFY (Receiving Variable Specified)

If a receiving numeric variable is included in a VERIFY statement, form #2 is assumed. In form #2, VERIFY continues checking sectors either until all sectors verify correctly, or until the first invalid sector is encountered. If all sectors verify successfully, the receiving variable is set equal to zero. As soon as an invalid sector is found, however, the VERIFY operation terminates at that point, and sets the receiving variable equal to the sector address of the next sequential sector (i.e., the first sector following the invalid sector). For example, if an error is discovered in sector #100, VERIFY action is halted, and the receiving variable is set equal to 101. No error message is displayed.

Because form #2 of VERIFY terminates at the first erroneous sector, the VERIFY operation must be resumed under program control at the next sequential sector (whose address is stored in the receiving variable) if the remaining sectors are to be verified. This form of VERIFY is particularly useful for diagnostic routines, because it permits the programmer to test for and react to the detection of an invalid sector under program control, without necessarily disturbing the current display on the CRT screen.

Example 2-48: Verifying a Specified Range of Sectors with VERIFY (Form #2) - Model 2260 and 2270 Series

```
500 VERIFY R (0, 1000) N
```

In line 500, 'N' is the specified receiving variable. VERIFY checks sectors #0 through #1000; if all 1001 sectors verify successfully, N = 0 at the conclusion of statement execution. If, however, a bad sector is discovered, 'N' is set equal to the address of the next sequential sector following the invalid sector, and the VERIFY operation is terminated at that point.

Example 2-49: Verifying a Specified Range of Sectors with VERIFY (Form #2) - Model 2280 Series

```
400 VERIFY T/D13, (0, 2500) E
```

In line 400, 'E' is the specified receiving variable. VERIFY checks sectors #0 through #2500 on the D13 fixed platter; if all 2501 sectors verify successfully, E = 0 at the conclusion of statement execution. If, however, a bad sector is discovered, 'E' is set equal to the address of the next sequential sector following the invalid sector, and the VERIFY operation is terminated at that point.

Like form #1, form #2 of VERIFY is frequently employed to verify the new catalog on the destination platter following a MOVE or COPY. As with form #1, form #2 automatically verifies the entire catalog if the starting and ending sector addresses are omitted from the VERIFY statement.

Example 2-50: Verifying the Catalog with VERIFY (Form #2) (Model 2260 and 2270 Series)

```
550 MOVE F TO R
560 VERIFY R N
```

Line 550 copies the catalog from the 'F' platter to the 'R' platter. Line 560 verifies the new catalog on the 'R' platter. Since no sector address parameters are specified, the entire catalog (starting at sector #0 and ending at the current end of the Catalog Area) is verified. 'N' is the receiving variable.

Example 2-51: Verifying the Catalog with VERIFY (Form #2) (Model 2280 Series)

```
440 MOVE T/D13, TO T/D10,
450 VERIFY T/D10, N
```

Line 440 copies the catalog from the D13 fixed platter to the D10 removable platter. Since no sector address parameters are specified, the entire catalog (starting at sector #0 and ending at the current end of the Catalog Area) is verified.

Note that the omission of starting and ending sector addresses from the VERIFY statement always implies that a catalog is to be verified; if the designated platter does not contain a catalog, either an error is signalled or arbitrary sectors may be verified.

CHAPTER 3
DISK DEVICE SELECTION AND MULTIPLE DATA FILES

3.1 INTRODUCTION

In Chapter 2 you were introduced to the most basic catalog procedures, including saving and loading programs and data files, skipping over records within a data file, scratching unwanted files, and moving the contents of the catalog from one platter to another. In the interests of simplicity and clarity of exposition, however, a number of important but complex disk operations were omitted from Chapter 2. Chapters 3 and 4 are therefore designed to expand and elaborate upon the discussion of catalog procedures begun in Chapter 2. Probably the most significant omission in that discussion was an explanation of how it is possible to keep more than one data file open on a disk at the same time. This subject is especially important because so many data processing problems involve the transfer of data from one file to another, or the storing of data in or reading of data from several different files in the course of processing transactions. Such operations would be time consuming in the extreme if each file had to be re-opened every time a record was to be written into it or read from it. Chapter 3 discusses the procedures for maintaining multiple open files on disk simultaneously. The related questions of how the disk is addressed, and how multiple disk units can be operated by a single system, also are examined in this chapter.

3.2 DISK DEVICE SELECTION

Chapter 2 presented you with what was, essentially, a "recipe" for using the disk. You were told that by executing a particular statement which included particular parameters, you could elicit a particular response from the system. The system itself remained a black box, however, whose internal workings were only vaguely hinted at. Although such an approach was appropriate for the purposes of Chapter 2, it cannot safely be followed in the present chapter. Some understanding of the internal operations of the system, particularly those which relate to management of the disk, is a necessary prelude to any discussion of how the system maintains open data files. The first topic to be considered is the mechanism by which the system is able to identify the disk unit and the individual platters within it.

Whenever a disk statement or command is executed, the system has immediate need for at least two items of information: the disk platter parameter 'F' 'R' or 'T' and a three-hexdigit device address. Together these items identify the disk unit and platter of the unit to be accessed. For certain disk statements and commands, the disk device address can, like the disk platter parameter ('F' 'R' or 'T'), be specified directly in the statement or command itself. For example, the statement

```
10 LOAD F /350, "PROG 1"
```

causes the system to access the disk unit with device address 350. On the Model 2270-3 and Model 2270A-3, this statement accesses platter #3. Likewise, on a Model 2280 the statement

```
10 LOAD T /D10, "PROG 1"
```

causes the system to access the removable platter designated by address D10.

In general, however, it is only necessary to specify the device address in a statement or command, when access to an address other than 310 is desired, since if no address is specified, the system automatically uses the default disk address, 310. The default address is stored by the system in a special section of system memory called the Device Table. Whenever a disk statement or command is executed, the system's first reaction is to check the Device Table for a disk device address (unless, of course, the address has been specified in the statement or command).

The Device Table

The Device Table in CPU memory is made up of 16 rows, or "slots" each of which is identified by a unique file number from #0 to #15 (see Figure 3-1 below). The Disk Device Address location of the default device address (310) is stored in the slot opposite #0. For this reason, #0 is referred to as the default file number.

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	STARTING SECTOR ADDRESS	CURRENT SECTOR ADDRESS	ENDING SECTOR ADDRESS
#0	0	3	10	00000	00000	00000
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	0	0	00	00000	00000	00000
#4	0	0	00	00000	00000	00000
#5	0	0	00	00000	00000	00000
.			.	.	.	.
.			.	.	.	.
.			.	.	.	.
#15	0	0	00	00000	00000	00000

Figure 3-1. The Device Table in Memory

Also in Figure 3-1, each of the 16 slots (#0-#15) has a location for the file status. The file status is used to indicate the status of the device table slot as follows:

0 = Data file not opened.

1 = Data file is open on a fixed platter (or diskette drive #1 or #3 in the Model 2270/70A Series) with a DATASAVE DC OPEN or DATALOAD DC OPEN statement.

2 = Data file is open on the removable platter (or diskette drive #2 in the Model 2270/70A Series) with a DATASAVE DC OPEN or DATALOAD DC OPEN statement.

3 = Data file is open on the platter specified by the low hexdigit of the disk unit device address (Model 2280 Series).

Adjacent to the file status is the device type (hexdigit) and the disk unit device address (two hexdigits). Each slot also has locations for three other items of information, a Starting Sector Address, a Current Sector Address, and an Ending Sector Address. The Device Table address parameters, used by the system to maintain open data files on disk, are discussed in the following section.

The default device address (310) which is a combination of the device type 3 and the unit address 10 are always stored next to the default file number (#0) by the system itself. Even after the system is Master Initialized (that is, the main power switch is turned OFF and then ON), thus clearing out all of memory, the system automatically returns 310 to its location opposite #0 in the Device Table (Figure 3-1).

For this reason, it is always possible to execute a disk statement or command without specifying a device address of 310. When, for example, a statement such as

```
10 LOAD F "PROG1"
```

is executed, the system automatically goes to the Device Table and checks for the default address opposite #0. Likewise, with a Model 2280, when a statement such as

```
10 LOAD T "PROG1"
```

is executed, the system automatically goes to the Device Table and checks for the default address opposite #0.

It is also possible, however, to store a device address in the Disk Device Address location opposite any one of the other file numbers (#1 - #15) in the Device Table. In this case, the device address must be stored in the table with a SELECT statement.

Example 3-1: Storing Disk Device Addresses In The Device Table

```
50 SELECT #3/310, #5/310
```

Statement 50 instructs the system to store disk device address 310 opposite file numbers #3 and #5 in the Device Table. Following the execution of statement 50, the Device Table looks like the table shown in Figure 3-2.

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	0	3	10	00000	00000	00000
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	0	3	10	00000	00000	00000
#4	0	0	00	00000	00000	00000
#5	0	3	10	00000	00000	00000
.			.	.	.	.
.			.	.	.	.
.			.	.	.	.
#15	0	0	00	00000	00000	00000

Figure 3-2. The Device Table with Disk Device Addresses Stored Opposite File Numbers #3 and #5

Notice that device address 310 is now stored in the Disk Device Address location opposite file numbers #3 and #5, as well as in the default slot (opposite #0). The file numbers #3 and #5 can now be used in a disk statement or command to reference device address 310 indirectly. For example, if a statement such as

```
60 LOAD F #3, "PROG2"
```

is now executed, the system immediately checks the Device Table for a device address opposite #3. Upon finding address 310, it proceeds to the disk unit and accesses the 'F' platter. If no address were stored opposite #3, or if the address of a device other than the disk (say, an interface board) were stored there, the system will signal an error when the disk statement is executed.

In summary, then, it is possible to specify a disk device address in two ways: directly (by including the address itself, or indirectly (by referencing a file number associated with the appropriate address). Therefore, a statement of the form

```
10 LOAD F /310, "PROG2"
```

is equivalent to the pair of statements

```
10 SELECT #3/310
20 LOAD F #3, "PROG2"
```

Note, however, that the data file manipulation statements (DATASAVE DC OPEN, DATASAVE DC, DATALOAD DC, etc.) do not permit the direct specification of a device address within the statement. In these statements, therefore, the device address must be referenced indirectly via a file number. This restriction is important because file numbers play a most critical role in the manipulation of cataloged data files.

Use Of File Numbers in Accessing the #3 Drive Model 2270-3/2270A-3 and Slave Drive Model 2260BC-2, 2260C-2, or Model 2280 Dual Drive

The #3 drive in the Model 2270-3 and the slave drive in the Model 2260BC-2, 2260C-2 or 2280 dual drive have a special device address, 350. This address can be stored in a slot opposite one of the file numbers #0 - #15 in the Device Table; subsequent reference to the associated file number then causes the system to access drive #3. For example, the statement

```
50 SELECT #2/350
```

causes device address 350 to be stored opposite #2 in the Device Table. A statement which references file #2, such as

```
60 LOAD F #2, "PROG1"
```

will now indirectly reference address 350, and access drive #3 to load in PROG 1 from the diskette mounted in that drive, *or* the F platter in the slave drive (whichever is at address 350- either a Model 2270-3/2270A-3 *or* a Model 2260BC-2/2260C-2). See Section 3.8.

Why Use the Device Table?

It may appear somewhat inefficient to use a section of memory and a special statement to store device addresses when the address can be supplied in the statement or command itself, or when, as in the normal case, no address need be supplied at all. If the Device Table were used exclusively to store device addresses, there would hardly be justification for belaboring the reader with an explanation of its purpose and operation. In fact, however, the Device Table serves a second and far more important function in connection with disk operations. The slots in the Device Table are utilized by the system to store critical information on currently open data files. Without the Device Table, therefore, it would not be possible to maintain multiple open files on the disk.

NOTE:

The Device Table slots #1 - #15 are used to store other device information as well as disk file information. A statement of the form SELECT #1/04C, for example, stores the interface board address 04C (with the 0 under the device type column and the 4C under the unit device address column) opposite file number #1 in the Device Table. If you are using disk and other devices in conjunction, therefore, be sure to use different file numbers for your disk and non-disk files.

3.3 MAINTAINING MULTIPLE OPEN DATA FILES ON DISK

The concept of an "open" data file was introduced in Chapter 2 with little exposition. It was pointed out simply that DATASAVE DC OPEN and DATALOAD DC OPEN are used to "open" and "re-open" a data file on disk; the actual procedures followed by the system in opening or re-opening a file were left undefined as internal operations.

This section details the specific and clearly defined procedure in opening a data file. In order to understand this procedure, however, you should first consider the kinds of information which the system requires in order to be able to access a file. Such information includes:

1. The disk platter and disk unit on which the data file is (or is to be) stored.
2. The starting sector address of the file.
3. The current sector address of the file (i.e., where the system is currently positioned in the file).
4. The ending sector address of the file.

Although some of this information (specifically, items 2 and 4) can be found in the Catalog Index, it is efficient for the system to have all of it at hand in one place. As you may already have suspected, that "one place" is the Device Table. The Device Table provides a convenient location in memory for the temporary storage of all information required by the system to access and maintain a cataloged data file. Such information is automatically copied from the Catalog Index into the Device Table whenever a data file is opened initially (with DATASAVE DC OPEN) or re-opened (with DATALOAD DC OPEN). In either case, the system first checks the default slot (or one of the other slots, #1 - #15, if a file number has been specified in the statement) for a valid disk address. If the slot contains no address, or an invalid address (for example, a tape address), an error is signalled and execution halts. If a valid address is found, the system proceeds to access the appropriate platter in the specified disk unit.

For an existing file which is re-opened with a DATALOAD DC OPEN statement, the system merely copies the file's starting and ending sector addresses from the Catalog Index into the default slot (or one of the other slots, if a file number has been used) in the Device Table. The current sector address is set equal to the starting sector address. For a file newly opened on disk with DATASAVE DC OPEN, the system first reserves space on the designated platter, and enters the file's name and sector parameters in the Catalog Index. Once this is done, the parameters are copied to a specified slot in the Device Table. Suppose, for example, that file DATFIL-1 is to be opened on the 'F' platter. Statement 10 below might be used:

```
10 DATASAVE DC OPEN F (100) "DATFIL-1"
```

One hundred sectors are reserved for DATFIL-1 on the 'F' platter. Assuming DATFIL-1 is the first file to be opened on this platter, and assuming that the Catalog Index occupies sectors 0-23, the Index entry for DATFIL-1 looks like this:

NAME	**TYPE**	**START**	**END**	**USED**	**FREE**
DATFIL-1	D	00024	00123	00001	00099

Once the Catalog Index has been appropriately updated, the sector address parameters for DATFIL-1 are immediately written to the default slot in the Device Table, which therefore looks like the table shown in Figure 3-3.

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	1	3	10	00024	00024	00123
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	0	0	00	00000	00000	00000
#4	0	0	00	00000	00000	00000
#5	0	0	00	00000	00000	00000
.			.	.	.	.
.			.	.	.	.
.			.	.	.	.
#15	0	0	00	00000	00000	00000

Figure 3-3. The Device Table with One File Open (DATFIL-1)

The parameters stored opposite #0 are those of DATFIL-1. The value '1' under 'file status' now indicates that a data file has been opened on a fixed platter. (Note that the current address has been set equal to the starting address at this point.) DATFIL-1 is now officially "open" and any DATASAVE or DATALOAD statement automatically accesses it.

Suppose, however, that a second file is now opened:

```
20 DATASAVE DC OPEN R (250) "DATFIL-2"
```

Execution of statement 20 causes the system to run through the same procedure followed in opening DATFIL-1, with the result that DATFIL-1's parameters opposite #0 in the Device Table are replaced with those of DATFIL-2. The Device Table appears as shown in Figure 3-4, following execution of statement 20:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	2	3	10	00124	00124	00323
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	0	0	00	00000	00000	00000
#4	0	0	00	00000	00000	00000
#5	0	0	00	00000	00000	00000
. . .			. . .	. . .	. . .	. . .
#15	0	0	00	00000	00000	00000

Figure 3-4. The Device Table with One File Open (DATFIL-2)

DATFIL-2 now becomes the only currently open file on disk. The 'file status' now shows a '2' to indicate that a data file has been opened on the 'R' platter. Any DATASAVE DC or DATALOAD DC statement now accesses DATFIL-2 instead of DATFIL-1. The question then arises: if every new file erases information on the previous file from the default slot, how is it possible to have more than one file open at once? The answer to this question is somewhat obvious: different slots in the Device Table can be used to open different data files. Since there are 16 slots in the Device Table, a total of 16 files can be open at the same time.

NOTE TO MVP USERS:

In a 2200 MVP system, each partition maintains its own Device Table; and subsequently each partition can open up to 16 files simultaneously.

You have already seen that the first thing the system does when a disk statement is executed is to check the Device Table for a disk device address. In the two examples just cited, only the default slot (opposite #0) was used for file information. As you know, the system itself automatically keeps the default address in that slot. Before you can use any of the other slots to open new files, however, you must store the disk device address in them with a SELECT statement such as the one illustrated in Example 3-1:

```
50 SELECT #3/310, #5/310
```

As you have already seen, this statement instructs the system to store disk device address 310 in the Device Table opposite File Numbers #3 and #5. The new Device Table is shown in Figure 3-5.

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	2	3	10	00124	00124	00373
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	0	0	00	00000	00000	00000
#4	0	0	00	00000	00000	00000
#5	0	0	00	00000	00000	00000
.			.	.	.	.
.			.	.	.	.
.			.	.	.	.
#15	0	0	00	00000	00000	00000

Figure 3-5. The Device Table with Disk Device Addresses Stored Opposite File Numbers #3 and #5, and One Open File (DATFIL-2)

The slots opposite #3 and #5 can now be used, in addition to the default slot, to store the sector address parameters of open files. To use one of these slots, it is necessary only to specify its file number in a DATASAVE DC OPEN or DATALOAD DC OPEN statement. Example 3-2 uses file #3 to open a second data file on the disk.

Example 3-2: Opening a New Data File with a File Number

```
150 DATASAVE DC OPEN F #3, (50) "DATFIL-3"
```

Statement 150 causes the system to check the slot opposite #3 for a device address. Upon finding address 310, the system goes to the disk unit and accesses the 'F' platter. Fifty sectors are reserved for DATFIL-3, and the file's name and location are entered in the Catalog Index. The file's sector address parameters (starting, ending, and current) are then written in the slot opposite #3 in the Device Table:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	2	3	10	00124	00124	00373
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	1	3	10	00374	00374	00423
#4	0	0	00	00000	00000	00000
#5	0	3	10	00000	00000	00000
. . .			. . .	. . .	. . .	. . .
#15	0	0	00	00000	00000	00000

Figure 3-6. The Device Table with Two Open Files

Obviously, the system must have some way of distinguishing DATFIL-2 from DATFIL-3 when data is to be stored in or retrieved from each file. Since the file names are not entered in the Device Table, the system can identify each file only by its associated file number. The file number associated with a file must therefore be used in any subsequent disk statement or command which accesses that file. The default file is, of course, automatically accessed if no file number is specified. Thus, the statement

```
160 DATASAVE DC A$( )
```

causes array A$( ) to be stored in DATFIL-2 (since DATFIL-2's parameters are stored opposite #0 in the default slot), while the statement

```
170 DATASAVE DC #3,A$( )
```

causes A$( ) to be saved in DATFIL-3 (since DATFIL-3's parameters are stored opposite #3).

Example 3-3: Referencing an Open File by File Number

```
10 SELECT #5/310
20 DATASAVE DC OPEN F #5, (50) "FIRST"
30 DATASAVE DC #5, A()
40 DATASAVE DC #5, END
```

Statement 10 writes the disk address (310) in the slot opposite #5 in the Device Table. Statement 20 opens FIRST and assigns its parameters to slot #5 in the Device Table. Statement 30 writes data from array A() into FIRST, and statement 40 writes an end-of-file trailer record to FIRST. Notice that both statements reference FIRST by specifying the file number (#5) to which it is assigned in the Device Table. When statements 30 and 40 are executed, the system immediately checks the slot opposite #5 in the Device Table for a disk address. It then accesses the specified disk and begins storing data at the sector specified in the Current Sector Address parameter of slot #5. Following execution of statement 40, the Device Table looks like Figure 3-7.

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	2	3	10	00124	00124	00373
#1	0	0	00	00000	00000	00000
#2	0	0	00	00000	00000	00000
#3	1	3	10	00374	00374	00423
#4	0	0	00	00000	00000	00000
#5	1	3	10	00424	00428	00473
.			.	.	.	.
.			.	.	.	.
.			.	.	.	.
#15	0	0	00	00000	00000	00000

Figure 3-7. The Device Table in Memory with Three Open Files

Existing files re-opened with DATALOAD DC OPEN also can be assigned file numbers. It is not required that a file be reassigned its original file number every time it is re-opened; the parameters of a file are copied anew into the Device Table each time it is re-opened, and it may be assigned any available file number. The file FIRST, opened initially in Example 3-3 above, might subsequently be re-opened and assigned a different file number, as illustrated in Example 3-4 below.

Example 3-4: Referencing an Open File by File Number

```
10 SELECT #4/310
20 DATALOAD DC OPEN F #4, "FIRST"
30 DSKIP #4, END
40 DATASAVE DC #4, B( )
50 DATASAVE DC #4, END
```

Statement 10 writes disk address 310 in the slot opposite #4 in the Device Table. Statement 20 opens an existing file, FIRST, and assigns its parameters to slot #4 in the Device Table. Statement 30 skips to the current end-of-file trailer record in the file, statement 40 saves a new record in the file from array B() over the trailer record, and statement 50 writes a new trailer record in the file. Notice that all reference to FIRST in statements 30, 40, and 50 is in terms of the file number (#4) to which it is assigned in the Device Table. Notice also that #4 is not the file number originally assigned to FIRST when it was initially opened in Example 3-3.

Note that it is possible to re-open the same file repeatedly, using a different file number each time. In this manner, every slot in the Device Table can be filled with the parameters of a single file. The practical advantage of such an arrangement would, however, be open to some doubt.

Using a Variable to Store the File Number

If it is convenient, a numeric variable can be used to store a file number. Subsequently, the variable can be included in a disk statement or command to reference the file number. For example, the statements

```
 5 SELECT #3/310
10 A = 3
20 DATALOAD DC OPEN F #A, "DATFIL-1"
```

cause the system to re-open DATFIL-1 on the 'F' platter, and store its parameters opposite #3 in the Device Table (since A=3). (Note that the use of numeric variables to reference file numbers is not legal in the SELECT statement itself. Thus, a statement of the form SELECT #A/310 is not permitted.)

3.4 THE "CURRENT SECTOR ADDRESS" PARAMETER

In the discussion of skipping over logical records within data files in Chapter 2, as well as in the recent discussion of storing data in a data file, you have seen why it is important, in fact necessary, for the system to know where it is positioned within a file at all times. If the system does not know, for example, that it has just stored a record ending at sector 86 in a currently open file, then it cannot know that the next record in that file must be saved starting at sector 87. In such a case, the system would obviously be incapable of maintaining data files on disk at all.

The system knows where it is positioned in a file by referring to the Current Sector Address of the file. The Current Sector Address is updated every time a record is saved in or loaded from a file, and every time records are skipped or backspaced over in a file. The Current Sector Address always indicates the next available sector following the most recent access of a file. For example, suppose that the file DATFIL 2 is to be saved on a fixed disk platter:

```
300 DATASAVE DC OPEN F #1, (500) "DATFIL-2"
```

The Device Table slot for DATFIL-2 now looks like this:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#1	1	3	10	00060	00060	00559

Figure 3-8. Device Table Slot for DATFIL-2

Notice that the Current Sector Address for DATFIL-2 is identical to the Starting Sector Address. This is the case whenever a file is opened or re-opened.

Suppose, now, that you store data from an array, A( ), into DATFIL-2:

```
DATASAVE DC #1, A( )
```

Assuming that the data from A( ) occupies one sector on disk, the Device Table slot for DATFIL-2 now reads as follows:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#1	1	3	10	00060	00061	00559

Current Address now updated to show that sector 61 is the next available sector.

Figure 3-9. Updated Device Table Slot for DATFIL-2

Notice that the Current Address is now updated to show that sector 61 is the next available sector in the file, since sector 60 (the first sector in the file) has been filled with data.

You might now save three more arrays of data:

```
310 DATASAVE DC #1, B( )
320 DATASAVE DC #1, C( )
330 DATASAVE DC #1, D( )
340 DATASAVE DC #1, END
```

Following execution of these statements (and assuming each array requires one sector on disk), the Device Table looks like this:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#1	1	3	10	00060	00064	00559

Figure 3-10. Updated Device Table Slot for DATFIL-2

Suppose, however, that you now want to skip back and load record #1. You use the DBACKSPACE statement:

```
350 DBACKSPACE #1, BEG
```

This statement instructs the system to set the value of the Current Sector Address equal to the value of the Starting Sector Address. Following execution of Statement 350, the Device Table looks like this:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#1	1	3	10	00060	00060	00559

Current Address now set back to address of first sector in file.

Figure 3-11. Updated Device Table Slot for DATFIL-2 Following Execution of a DBACKSPACE BEG Statement

At this point, you can load the first record from DATFIL-2.

3.5 CLOSING A DATA FILE

You should understand more clearly now the precise meanings of the concepts of "opening" and "closing" a data file. A data file is opened (by a DATASAVE DC OPEN or DATALOAD DC OPEN statement) when its parameters are entered in a slot in the Device Table. A data file is closed when its parameters are removed from the Device Table, either by writing over the parameters with another set of parameters, or by zeroing out the parameters. There are four methods of closing a currently open data file:

1. Assigning the file number currently associated with the file to another file.
2. Executing a CLEAR command with no parameters.
3. Master Initializing the system.
4. Executing a DATASAVE DC CLOSE statement.

Each of these four methods is explained in the following paragraphs.

1. Assigning the file number currently associated with the file to another file. This operation causes the parameters of the new file to be written over the parameters of the original file, thus closing the original file.

Example 3-5: Closing a Data File by Re-Assigning Its File Number

```
110 SELECT #1/310
120 DATASAVE DC OPEN F #1, (110) "DATFIL-1"
150 DATASAVE DC OPEN R #1, (600) "DATFIL-2"
```

Statement 110 selects file number #1 to the disk unit address 310. Statement 120 opens DATFIL-1, reserves 110 sectors for it on the 'F' disk platter, and causes its parameters to be entered in the Device Table in the slot opposite #1. Statement 150 opens a new data file, DATFIL-2, and stores its parameters in slot #1. These parameters overwrite those of DATFIL-1, effectively closing DATFIL-1.

2. Executing a CLEAR command with no parameters. A CLEAR command with no parameters causes all of memory to be cleared, including the Device Table. All the information in the Device Table is zeroed out, thereby closing all files (the #0 disk address is maintained).

3. Master Initializing the system (i.e., throwing the CPU power switch OFF and then ON). This, too, has the effect of clearing out memory, thus closing all files.

4. Executing a DATASAVE DC CLOSE statement. DATASAVE DC CLOSE should not be confused with DATASAVE DC END. DATASAVE DC END causes an end-of-file trailer record to be written in the specified file. DATASAVE DC CLOSE, however, causes all sector address parameters for the specified file or files in the Device Table to be zeroed out, thereby closing the file(s). The disk device address stored in a slot is not zeroed out by DATASAVE DC CLOSE, however.

Example 3-6: Closing a Specified File with a DATASAVE DC CLOSE Statement

```
200 DATASAVE DC CLOSE
210 DATASAVE DC CLOSE #1
```

Statement 200 causes the sector address parameters associated with the default file #0 (since no file number is specified) to be zeroed out, thus closing the file associated with #0. Statement 210 causes the sector address parameters stored in slot #1 to be zeroed out, thus closing the file associated with #1.

Example 3-7: Closing All Currently-Open Files with a DATASAVE DC CLOSE Statement

```
300 DATASAVE DC CLOSE ALL
```

Statement 300 causes all sector address parameters in the Device Table to be zeroed out, thus closing all currently open files.

It is generally good practice to close a data file once processing is complete. In this way, you can prevent someone else from accidentally saving data into the file over currently stored data, and thus destroying your data. You should also always be sure to write a data trailer record in the file prior to closing it, since you will then be able to re-open the file, skip to the end, and continue storing data in it at a later date.

When a file is closed, by whatever method, its three sector address parameters are removed from the Device Table. When the file is subsequently re-opened with a DATALOAD DC OPEN statement, the Current Sector Address is automatically set equal to the Starting Sector Address.

3.6 SKIPPING AND BACKSPACING OVER INDIVIDUAL SECTORS IN A FILE

In Chapter 2, the discussion of DSKIP and DBACKSPACE was confined to the skipping of logical records within a file. It is also possible, however, to skip individual sectors in a file. This method is a much faster way of moving through a file than skipping records, but its value cannot be fully understood until the process of skipping logical records is examined in greater detail.

Remember that a logical record may consist of any number of sectors. The first logical record in a file might, for example, contain three sectors, while the second contains thirteen. The system has no way of knowing in advance how many sectors are in each record; when the system is instructed to skip or backspace over a prescribed number of records, therefore, it must actually read those records from the specified file and update the Current Sector Address after the specified number of records have been read. Suppose, for example, that the system is currently positioned at the beginning of DATFIL-1, and that DATFIL-1 is associated with file #1 in the Device Table. If you want to skip three records in DATFIL-1, you would execute a DSKIP #1,3 statement. Such a statement causes the system to run through the following set of operations:

1. Check the Current Sector Address in slot #1 in the Device Table to see where it is currently positioned in the file.
2. Access the disk and read three logical records, beginning at the location specified in the Current Sector Address parameter.
3. After reading the third logical record, check the sector address of the last sector in that record.
4. Set the Current Sector Address in slot #1 equal to one greater than the address of the last sector in logical record #3. This is the address of the next available sector in the file following record #3.

At the end of this procedure, the Current Sector Address in slot #1 is equal to the address of the first sector of record #4 (or the first available sector following record #3, if record #4 has not yet been saved).

Suppose, now, that you know there are three sectors in each logical record in DATFIL-1. In this case, if you want to skip three logical records, you can simply instruct the system to skip nine sectors. Since the system knows exactly how many sectors are to be skipped, it need not access the disk and read the records themselves. It simply increments the Current Sector Address by nine, and records this value in slot #1 as the new Current Address. The process of skipping or backspacing through a file is greatly accelerated, since no disk accesses are required.

The 'S' parameter is used in a DSKIP or DBACKSPACE statement to inform the system that it is to skip a specified number of sectors rather than logical records.

Example 3-8: Skipping over a Number of Sectors in a File

```
400 DSKIP #1, 20S
```

Statement 400 instructs the system to increment the Current Address for the file associated with slot #1 in the Device Table by 20. If the old Current Address was equal to X, the new Current Address is equal to X+20. If each logical record consists of four sectors, this statement has the effect of skipping over five logical records.

Example 3-9: Backspacing over a Number of Sectors in a File

```
410 DBACKSPACE #3, 25S
```

Statement 410 instructs the system to decrement the Current Address for the file associated with #3 in the Device Table by 25. If the original Current Address was equal to Y, the new Current Address is equal to Y-25. If each logical record consists of five sectors, this statement has the effect of backspacing over five logical records.

It is important, however, that every logical record in the file consist of the same number of sectors; otherwise, skipping or backspacing over a number of sectors can lead to serious problems. If you do not know exactly how many sectors to skip, you may end up somewhere in the middle of a logical record. Since you normally want to read an entire logical record rather than some arbitrary portion of one, that could prove disastous.

3.7 THE "T" PLATTER PARAMETER IN DISK OPERATIONS

Until now, only two parameters have been discussed in connection with accessing a disk platter on the Model 2260 and 2270 series disk systems - the 'F' parameter and the 'R' parameter.* These parameters are "absolute" in the sense that each identifies a single disk platter. The reference of each parameter is fixed and cannot be changed (that is, the 'F' parameter can never be used to access the 'R' platter, and vice versa).

Such an arrangement lacks some flexibility. It is desirable in certain programming cases to be able to access either the 'F' or 'R' platter with the same disk statement or command. The 'T' parameter provides such a capability. When the 'T' parameter is specified in a disk statement or command instead of 'F' or 'R' it causes the system to use the disk device address to determine which platter is to be accessed, and access the designated platter.

* The 'F' and 'R' parameters have a limited application with the Model 2280 disk systems. An F/310, R/310 and a T/B10 can be used to access the first two platters of the Model 2280.

For such a technique to be possible, however, each disk platter must have its own device address. This is true only in a very limited sense. The disk device address (e.g., "310") is really a conjunction of a device type and a unit device address. The first hexdigit of the disk address is the device type; the remaining two hexdigits form the unit device address. It is the device type which can be used to designate a particular disk platter.

3	10
↑	↑
Device Type	Unit Device Address

In all of the examples to this point, a single device type, "3" (e.g., 310, 320, etc.), has been used consistently. However, a second device type, "B" (e.g., B10, B20, etc.), is also permissible in a disk device address. When used in conjunction with the 'T' parameter, a device type of "3" designates the 'F' disk platter, while a device type of "B" designates the 'R' platter:

3	10	B	10
↑	↑	↑	↑
Designates the 'F' platter when 'T' parameter is used.	Designates the primary disk drive.	Designates the 'R' platter when 'T' parameter is used.	Designates the primary disk drive.

For example the statement

```
10 LOAD T/310, "PROG1"
```

causes the system to access the 'F' platter, while the statement

```
20 LOAD T/B10, "PROG2"
```

causes the system to access the 'R' platter. It should be emphasized that a disk device address is never used by itself to access a disk platter; it is always necessary to specify one of the parameters 'F' 'R' or 'T' in statements where such a parameter is required.

No mention was made of the "B" device type in previous examples because the device type "B" itself is significant only when the 'T' parameter is specified. The 'F' or 'R' parameter, when specified, always overrides the device type. Thus, for example, the command

```
LOAD F/310, "PROG1"
```

access the 'F' platter; and so too does the command

```
LOAD F/B10, "PROG1".
```

In this case, the device type ("B") has no meaning to the system other than to indicate a disk device.

SPECIAL NOTE TO MODEL 2270-1/2270A-1 AND SINGLE MINIDISKETTE OWNERS:

An address of B10 is illegal when used in conjunction with the 'T' parameter, when the disk unit contains only a single disk drive, which must be referenced as an 'F' platter (i.e., with address 310). Otherwise, an error results.

SPECIAL NOTE TO MODEL 2270-3/2270A-3 OWNERS:

When the 'T' parameter is used with the Model 2270-3, platter selection is determined by both the device type and the device address. If the device address is the primary address (i.e., X10, X20, X30, etc.), then a device type of "3" designates Platter #1 (i.e., 310, 320, 330, etc.), while a device type of "B" designates Platter #2 (i.e., B10, B20, B30, etc.). If, on the other hand, the device address is not the primary address (e.g., 350, 360, 370, etc.), then Platter #3 is accessed. Note that the addresses B50, B60, and B70 are not legal in this case, since the diskette in drive #3 must be designated as an 'F' platter.

The 'T' parameter provides maximum flexibility when used in a statement which references a file number specified as the value of a variable. In such a case, the system determines the specified file number from the value of the variable, and then checks the Device Table and inspects the device type of the device address stored opposite the specified file number. This arrangement makes it possible to use the same disk statement to access all platters in the system simply by changing the value of the file number variables.

Example 3-10: Accessing More Than One Disk Platter with the 'T' Parameter

```
10 SELECT #3/310, #4/B10
:
:
100 A = 3: B$ = "DATFIL": GOSUB 200
:
:
200 DATALOAD DC OPEN T #A, B$
210 RETURN
```

Statement 10 stores disk device addresses 310 and B10 in slots #3 and #4 of the Device Table, respectively. Subsequently, statement 100 sets A equal to 3, and B$ equal to "DATFIL" and branches to a subroutine located at line 200. The subroutine re-opens DATFIL on the 'F' platter (since A = 3, #A = #3 and disk address 310 is stored opposite #3).

The same subroutine could be used to access the 'R' platter if called from another point in the program:

```
150 A = 4: B$ = "TEST-2": GOSUB 200
:
:
200 DATALOAD DC OPEN T #A, B$
210 RETURN
```

In this case, data file TEST-2, located on the 'R' platter, is re-opened by the subroutine.

The 'T' parameter provides the general capability to write disk statements which can access any disk platter. This feature may prove particularly useful for file update operations, where the same file may reside alternately on different platters. Users of the Model 2260 series should find the 'T' parameter helpful in debugging file maintenance programs written for the fixed platter by testing them with dummy files stored on the removable platter (thus avoiding the danger of erasing legitimate data on the fixed platter). Finally, owners of Models 2270-3, 2270A-3, 2260C-2, and 2260BC-2 will find the 'T' parameter helpful because it provides them with a single parameter which can be used to access all the disk platters. For example, a program can be designed which makes a specific platter (and disk unit) selectable by the operator when the program is run:

Example 3-11: Using the 'T' parameter to Access a User-Selectable Disk Platter

```
10 INPUT "ENTER PLATTER-NUMBER (1,2, OR 3)" A
20 INPUT "ENTER PROGRAM NAME" N$
30 ON A SELECT #1/310; #1/B10; #1/350: ELSE GOTO 10
40 LOAD T #1, N$
```

Device Type 'D'

A third device type 'D' which is always used with the 'T' platter parameter, can be used for disk devices. Device type 'D' indicates that the platter to be accessed is specified by the third hexdigit of the device address. For example, the device address D12 is interpreted as:

D	1	2
↑	↑	↑
device type	unit device address (10)	platter number

Note that when type 'D' is used, the low hexdigit of the device address is implied to be 0. Platter numbers greater than 1 are used only on Model 2280 disk drives. For other disk drives, T/D10 is equivalent to R/310 or T/B10, and T/D11 is equivalent to F/310 or T/310.

Changing The Default Address

It is possible to change the default disk address (normally 310) and reference the new default address via the 'T' parameter.

The disk default address (310) is changed with a SELECT DISK statement. For example, the statement

```
50 SELECT DISK/B10
```

causes disk address B10 to be recorded in slot #0 in the Device Table:

FILE NUMBER	FILE STATUS	DEVICE TYPE	DISK UNIT DEVICE ADDRESS	START	CURRENT	END
#0	0	B	10	00000	00000	00000

Figure 3-12. The Device Table Following Execution of a SELECT DISK/B10 Statement

Once statement 50 is executed, any disk statement or command containing the 'T' parameter with no file number specified causes the system to inspect the device type in the new default address (B10) and access the 'R' disk platter. Note that the default address can also be changed with a statement of the form SELECT #0/B10.

Example 3-12: Using the 'T' Parameter with a New Default Address

```
10 SELECT DISK/B10
20 DATASAVE DC OPEN T (100) "DATFIL-1"
```

Statement 10 changes the default address (stored in slot #0) from 310 to B10. Statement 20 causes the system to check the default address in the default slot (since no file number is specified) and, since the 'T' parameter is used, to inspect the device type in the address. In this case, the device type is B (B10); the 'R' platter is therefore used to open DATFIL-1.

After it has been changed, the default address can be reset to 310 by:

1. Entering a SELECT DISK/310 statement, or
2. Master Initializing the system.

3.8 MULTIPLE DISK UNITS

If you have only one disk unit attached to your system, the problem of multiple disk addresses does not concern you, since you will deal exclusively with the primary disk drive addresses 310 and B10 (and 350, on the Models 2270-3 and 2270A-3). Many installations, however, drive two or more disks with a single system. (A typical configuration includes one large fixed/removable disk drive for the data base, and a smaller diskette drive for software.) In multiple-disk configurations, the system distinguishes different disk units by means of the last two digits in their unit device addresses.

3/B	10
↑	↑
Device Type	Unit Device Address (Identifies disk drive number one)

Models 2260BC, 2260C, 2270/2270A-1, 2270/70A-2 and Minidiskette

For disk Models 2260B, 2260C, 2260BC, 2270-1, 2270-2, 2270A-1, 2270A-2, and Minidiskette, the unit device address of each successive disk unit on the same system is usually computed by adding HEX(10) to the disk device address of the primary disk. The addresses of successive disks are listed in Table 3-1.

Table 3-1. Disk Addresses for Models 2260C, 2260BC, 2270-1, 2270-2 and Minidiskette

Disk Unit no. 1	310 or B10
Disk Unit no. 2	320 or B20
Disk Unit no. 3	330 or B30

Models 2270-3 and 2270A-3

For the Models 2270-3 and 2270A-3, the addressing scheme is somewhat different. The unit device address of drives #1 and #2 in a second and third disk unit on the same system is computed by adding HEX(10) to the primary disk address (310). The address of the drive #3 is computed by adding HEX(40) to the primary disk address.

Table 3-2. Disk Addresses for Models 2270-3, and 2270A-3

	Drives #1 and #2	Drive #3
Disk Unit no. 1	310 or B10	350
Disk Unit no. 2	320 or B20	360
Disk Unit no. 3	330 or B30	370

Models 2260BC-2/2260C-2

For the Models 2260BC-2 and 2260C-2, the individual disks are addressed as shown in Table 3-3. The master drive in combination number 2 and combination number 3 of the same system is computed by adding HEX(10) to the primary disk address (310). Similarly, the address of the slave drive in the first three combinations is computed by adding HEX(10) to the previous address (350).

Table 3-3. Disk Addresses for Models 2260C-2/2260BC-2

	Master	Slave
Combination no. 1 (Primary)	310 or B10	350 or B50
Combination no. 2	320 or B20	360 or B60
Combination no. 3	330 or B30	370 or B70

Model 2280 Dual Drives

For the Model 2280 Dual Drives, the individual platters are addressed as shown in Table 3-4. The platter addresses in the master and slave drives of dual drives no. 2 and no. 3 of the same system are computed by adding HEX(10) to the corresponding platter address of dual drive no. 1. In the same dual drive, the slave platter address is computed by adding HEX(40) to the corresponding master platter address.

Table 3-4. Platter Addresses for Model 2280 Dual Drives

	Master	Slave
Dual Drive no. 1		
Removable	D10 (or B10)	D50 (or B50)
First Fixed	D11 (or 310)	D51 (or 350)
Second Fixed	D12	D52
Third Fixed	D13	D53
Fourth Fixed	D14	D54
Fifth Fixed	D15	D55
Dual Drive no. 2		
Removable	D20 (or B20)	D60 (or B60)
First Fixed	D21 (or 320)	D61 (or 360)
Second Fixed	D22	D62
Third Fixed	D23	D63
Fourth Fixed	D24	D64
Fifth Fixed	D25	D65
Dual Drive no. 3		
Removable	D30 (or B30)	D70 (or B70)
First Fixed	D31 (or 330)	D71 (or 370)
Second Fixed	D32	D72
Third Fixed	D33	D73
Fourth Fixed	D34	D74
Fifth Fixed	D35	D75

NOTE ON ALL DISK UNITS:

The device addresses for disk units are set at the factory, or by your Wang Service Representative. The address of each disk unit should be marked on the disk controller board for that unit. If you have questions about addressing multiple disks in a system, contact your Wang Service Representative.

Accessing Multiple Disk Units

The techniques for accessing a disk platter with catalog procedures are the same for additional disk units on a system as for the primary unit. A platter can be accessed in four ways:

1. Specifying the disk device address in a disk statement or command, e.g.:

   ```
   100 LOAD R /330, "PROG-1"
   ```

   Statement 100 loads PROG-1 from the 'R' platter in disk unit number three. Note that there are a number of catalog statements in which the device address cannot be directly specified.

2. By selecting a disk address as the default disk address, and referencing the default address, e.g.:

   ```
   100 SELECT DISK/032
   120 DATASAVE DC OPEN T (100) "DATFIL-1"
   ```

   Statement 100 changes the default address from 310 to D32, and statement 120 opens DATFIL-1 on the second fixed platter of a Model 2280 disk unit.

3. By assigning the disk address to a file number in the Device Table, and referencing the address indirectly, via the file number, e.g.:

   ```
   100 SELECT #3/320
   110 DATASAVE DC OPEN F #3, (100) "DATFIL-1"
   ```

   Statement 100 stores disk address 320 in the #3 slot in the Device Table, and statement 110 opens DATFIL-1 on the 'F' platter of disk unit number two. In this case, the disk unit is determined from the disk address, while the disk platter is specified in the DATASAVE DC OPEN statement ('F'). Alternatively, both the disk unit and the disk platter can be determined from the device address:

   ```
   100 SELECT #3/320
   110 DATASAVE OPEN T #3, (100) "DATFIL-1"
   ```

In this case, both the disk unit (number two) and the disk platter ('F' platter) are determined by inspection of the device address.

4. By assigning the device address to a file number in the Device Table, and referencing the file number indirectly (via a variable), e.g.:

   ```
   100 SELECT #3/B20
   105 A = 3: B$ = "DATFIL-1"
   110 DATASAVE DC OPEN T #A, (100) B$
   ```

   Since A = 3, and address B20 is stored in slot #3 in the Device Table, the file DATFIL-1 is opened on the 'R' platter of disk unit number two.

3.9 SUMMARY OF DEVICE TABLE ITEMS

File Status

Established only by:	DATASAVE DC OPEN or DATALOAD DC OPEN File status = 1 when data file is open on a fixed platter. File status = 2 when data file is open on a removable platter. File status = 3 when data file is open using device type 'D'; platter is specified by the low hexdigit of the disk unit address.
Cleared by:	DATASAVE DC CLOSE CLEAR LOAD RUN Master Initialization File status = 0, signifying the data file is not open.

Device Type and Address

Default (#0) set by:	SELECT #0
	SELECT DISK
	Master Initialization

#1 - #15 set by:	SELECT # n
Cleared by:	CLEAR LOAD RUN Master Initialization

File Start, Current, End

Established by:	DATALOAD DC OPEN DATASAVE DC OPEN
Modified by:	DATALOAD DC DATASAVE DC DSKIP DBACKSPACE

Cleared by:	CLEAR LOAD RUN DATASAVE DC CLOSE Master Initialization

NOTE:

The contents of the Device Table can be displayed in hexadecimal notation with a LIST DT command. (See the Wang BASIC-2 Language Reference Manual.)

CHAPTER 4
EFFICIENT USE OF THE DISK

4.1 INTRODUCTION

This chapter discusses several techniques designed to help you make more efficient use of your disk, both in terms of optimizing the use of disk storage space and speeding up processing time for disk files. The following topics are covered in the chapter:

1. Reserving additional space in program files for program expansion.
2. Establishing temporary work files on the disk.
3. Renaming and reusing scratched files.
4. Efficient use of disk storage space within records.
5. The LIMITS Statement.

4.2 PROGRAM FILES REVISED

The discussion of saving program files in Chapter 2 restricted itself to cases in which the system used exactly enough disk space to hold the recorded program lines. In many cases, however, it is advantageous to reserve additional sectors within a program file for future expansion of the program. If such additional space is reserved at the outset, the program can subsequently be expanded and written back into its original location in the catalog (the reuse of scratched program file locations is described in Section 4.5). If extra space is not reserved when the file is initially created, the expanded program may not fit into its original space, and must be saved at a new location in the Catalog Area. In this case, the space occupied by the old program is wasted, unless a new file can be found to occupy it. The SAVE command provides a means of reserving extra sectors in a program file when the program is initially stored on disk.

In order to reserve extra sectors in a program file, the number of additional sectors to be reserved must be enclosed in parentheses and listed in the SAVE DC command immediately before the program name. The system then automatically adds the specified number of additional sectors at the end of the program file when the program is recorded on disk.

Example 4-1: Reserving Additional Sectors in a Program File

```
SAVE F (10) "PROG-1"
```

This command instructs the system to record all program lines currently in memory on the 'F' disk platter, and name the file "PROG-1". In addition to the sectors needed to hold the program itself, 10 sectors are reserved for future additions to the program (see Figure 4-1).

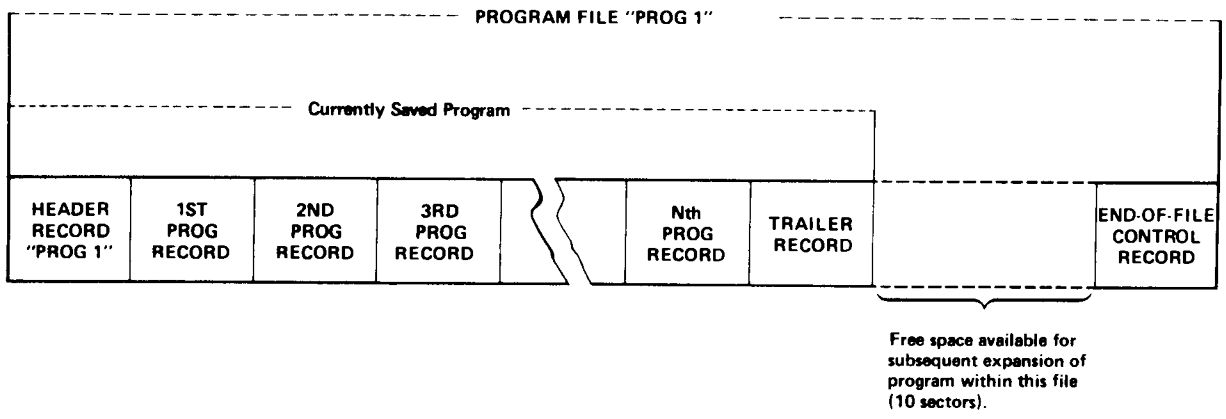


Figure 4-1. The Program File PROG-1 with Ten Extra Sectors Reserved

4.3 ESTABLISHING TEMPORARY WORK FILES ON DISK

Temporary work files can be useful in a variety of data processing operations. A "temporary" work file is opened with a DATASAVE DC OPEN statement, but unlike a regular cataloged file, it is not listed in the Catalog Index, and not stored in the Catalog Area on disk. Its parameters are, however, entered in the Device Table in memory. Temporary files may be used as transaction files, to contain transactions saved over a period of time and processed as a batch, or as scratch files, in which the results of intermediate calculations are stored prior to final processing. They may, in short, be used as a storage area for any type of transient data which is not sufficiently final to warrant storage in a permanent file.

Because they are not cataloged, temporary files must be stored outside the Catalog Area on disk. The end of the Catalog Area (that is, the address of the last sector reserved for the Catalog Area) is specified in the SCRATCH DISK statement when the catalog is established. If temporary files are to be used, the catalog may not occupy the entire platter; a number of sectors must be left outside the Catalog Area for the temporary files. For example, the Model 2260-B1/4 and 2260-C1/4 each have 4800 sectors on each platter. Since sector numbering starts at zero rather than one, the highest sector address is 4799. If a number of sectors (say, 100) are to be left available for temporary files, the address of the last sector in the Catalog Area must be 4799 minus 100, or 4699:

```
100 SCRATCH DISK F END=4699
```

Sectors 4700 through 4799 are left outside the Catalog Area, and may be used for temporary files.

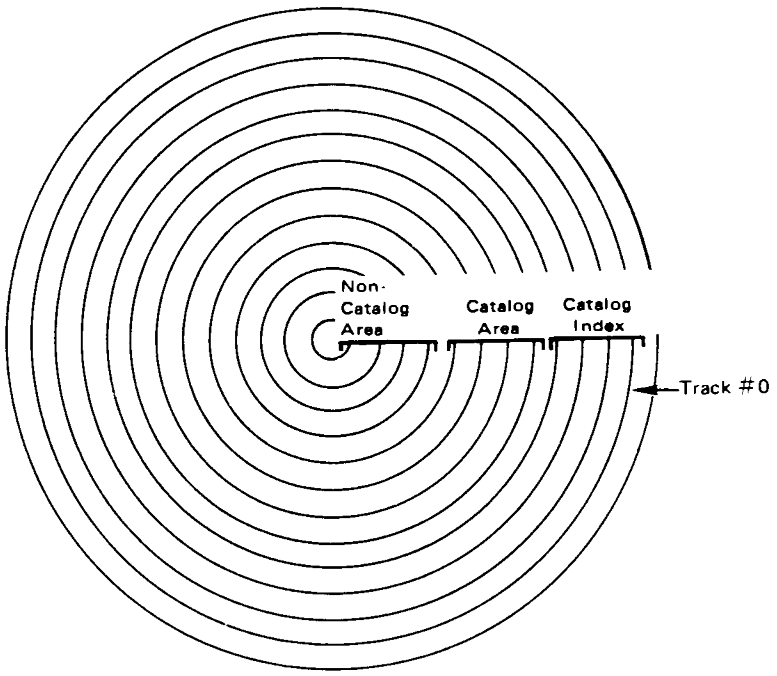


Figure 4-2. Layout of the Platter Surface Showing Catalog Index, Catalog Area, and Non-Catalog Area (Used for Storage of Temporary Files)

Temporary files are opened and accessed with the same BASIC statements used to open and access cataloged files. However, temporary files cannot be named, nor can they be accessed by name. Instead, the special TEMP parameter, along with the beginning and ending sector addresses of the temporary file, must be specified in the DATASAVE DC OPEN statement when the file is opened initially, and again in the DATALOAD DC OPEN statement when the file is re-opened.

Example 4-2: Opening a Temporary Work File on Disk

```
300 DATASAVE DC OPEN R TEMP 4700, 4799
```

Statement 300 opens a temporary work file on the 'R' disk platter. Sectors 4700 through 4799 are reserved for this temporary file (these sectors must be outside the Catalog Area). No information on the file is entered in the Catalog Index; however, the temporary file's parameters are entered in the default slot (#0) in the Device Table. Following the execution of statement 300, any DATASAVE DC or DATALOAD DC statement which does not specify a file number (i.e., which references the default slot) will read or write data in the temporary file.

Like cataloged files, temporary files can be assigned file numbers. In this way, more than one temporary file can be open at the same time.

Example 4-3: Opening More Than One Temporary Work File

```
300 SELECT #1/310, #3/310
320 DATASAVE DC OPEN F #1, TEMP 4700, 4749
330 DATASAVE DC OPEN F #3, TEMP 4750, 4799
```

Statement 300 stores disk address 310 opposite file numbers #1 and #3 in the Device Table. Statement 320 opens a temporary file on the 'F' platter, reserves sectors 4700 through 4749 for that file, and enters the file parameters in slot #1 of the Device Table. Statement 330 opens a second temporary file on the 'F' platter, occupying sectors 4750-4799, and assigns its parameters to slot #3 in the Device Table. Any reference to #1 or #3 in a DATASAVE DC or DATALOAD DC statement accesses these temporary files.

Data is stored in a temporary file just as it is stored in a cataloged file. As with a cataloged file, a data trailer record should always be written in the file at the completion of a data storage operation. As with cataloged data files, the last sector of a temporary data file is used by the system for control information, and therefore, at least one additional sector should be reserved for this purpose.

A temporary file is closed in the same way a cataloged file is closed; it is re-opened with a DATALOAD DC OPEN statement. The TEMP parameter and the beginning and ending sector addresses of the file must be specified.

Example 4-4: Re-opening a Temporary Work File

```
500 DATALOAD DC OPEN F TEMP 4750, 4799
```

Statement 500 re-opens an existing temporary file beginning at sector 4750 on the 'F' disk platter.

WARNING:

Temporary files must be used carefully since the system does not guard against opening a temporary file in the same location as an existing temporary file. It is the programmer's responsibility to assure that this does not occur. In a multi-user system (MVP or multiplexed 2200's) controlling temporary files may be difficult and is generally not recommended.

4.4 ALTERING THE CATALOG AREA

The upper limit of the Catalog Area is originally set with the END parameter in a SCRATCH DISK statement when the catalog is created. If more room is needed for temporary files, or if more sectors must be devoted to cataloged files, the size of the Catalog Area can be changed with a MOVE END statement. In this statement, it is necessary to specify only the sector address which is to become the new ending sector address of the Catalog Area. Note that MOVE END alters the size of the Catalog Area only; it does not change the size of the Catalog Index.

Example 4-5: Changing the Size of the Catalog Area

```
100 SCRATCH DISK F LS=30, END=4699
:
:
500 MOVE END F = 4599
```

Statement 100 sets the limit of the Catalog Area at sector 4699. Statement 500 moves the limit back 100 sectors, to sector 4599, thereby allowing 100 additional sectors to be used for temporary files (outside the Catalog Area). The Catalog Area may be expanded as well as constricted, but its upper limit must never exceed the highest sector address available on a disk platter. The size of the Catalog Index cannot be changed with MOVE END.

4.5 RENAMING AND REUSING SCRATCHED FILES

Temporary files offer one good way to make the most efficient use of disk storage space. Another way to get maximum use out of available disk storage area is to reuse the space occupied by scratched files. As you saw in Chapter 2, one way to eliminate scratched files is to execute a MOVE operation, since MOVE automatically deletes scratched files when it copies the catalog to a new platter. In many cases, however, it is easier and more efficient to store a new program or new data file directly into space occupied by a scratched file, without moving the whole catalog to a second platter. This is true particularly in the case of revised programs. New files are recorded in the space occupied by scratched files with the SAVE DC command and DATASAVE DC OPEN statement. The file type of the scratched file (program or data) is irrelevant when opening a new file in its space: a program file may be saved in the space occupied by a scratched data file, and a data file may be saved in the space occupied by a scratched program file. The scratched file name must precede the new file name in the SAVE DC command or DATASAVE DC OPEN statement.

Example 4-6: Saving a Program in Space Occupied by a Scratched File

```
SCRATCH R "PROG1"
SAVE R ("PROG1") "PROG2" 200, 500
```

The SCRATCH statement causes program file PROG 1 to be set to a scratched status. SAVE then stores the new program in lines 200 through 500 in the sectors previously reserved for PROG1, and names it "PROG2". The new file name ("PROG2") and location are entered in the Catalog Index. The scratched entry for PROG1 remains in the Catalog Index, although it no longer appears in a listing of the Index.

Notice that the scratched file name must be enclosed in parentheses when it is referenced in a SAVE command.

Example 4-7: Opening a Data File in Space Occupied by a Scratched File

```
10 SCRATCH F "DATFIL1"
20 DATASAVE DC OPEN F ("DATFIL1") "DATFIL2"
```

Statement 10 scratches DATFIL1. Statement 20 assigns the sectors previously reserved for DATFIL1 to DATFIL2, and updates the Catalog Index accordingly. DATFIL2's parameters (previously those of DATFIL1) are entered in the default slot (#0) in the Device Table. The scratched entry for DATFIL1 remains in the Catalog Index, although it is disabled and no longer appears in a listing of the Index.

A program file which has been scratched can be reused as a data file, and vice versa.

Example 4-8: Opening a Data File in Space Occupied by a Scratched Program File

```
10 SCRATCH F "PROG-1"
20 DATASAVE DC OPEN F #1, ("PROG-1") "DATFIL-3"
```

Statement 10 scratches PROG-1. Statement 20 assigns the sectors on disk previously reserved for PROG-1 to DATFIL-3, and updates the Catalog Index accordingly. DATFIL-3's parameters (previously those of PROG-1) are entered in slot #1 in the Device Table (the disk device address must previously have been stored opposite #1). The scratched entry for PROG-1 is not removed from the Catalog Index, however, although it no longer appears in the Index listing.

It is entirely possible to rename a scratched file with the same name. This feature is useful for revising program files, since the program can be updated and then resaved into the original location with the same name (assuming, of course, that additional space has been reserved in the original file for expansion of the program).

Example 4-9: Renaming a Scratched Program File with the Same Name

```
SCRATCH R "PROG-1"
SAVE R()"PROG-1"
```

The SCRATCH statement scratches PROG-1. The SAVE command subsequently resaves an updated version of the program, assigning it the same name ("PROG-1"), and storing it in the same location as the original PROG 1. If there is not enough space in the file for the new program, an error is signalled.

If a program is to be updated but the new program file is larger than the existing file, the following technique can be used. Assume "PROG" is the program file to be updated. "PROG" must be eliminated from the catalog index (by renaming the file) so that a new file can be created with the name "PROG".

Example 4-10: Updating a Program File with the Same Name

```
10 SCRATCH F "PROG"
20 SAVE F ("PROG") "JUNK" 0, 0
30 SCRATCH F "JUNK"
40 SAVE F (10) "PROG"
```

Statement 10 scratches PROG. Statement 20 retains the sectors previously reserved for PROG and names the new program JUNK. Since JUNK is created only to allow name "PROG" to be reused, statement 30 scratches JUNK. In statement 40 PROG is saved with extra space reserved so that future updating can be performed without renaming the file.

Finally, it is also possible to scratch and rename a data file without disturbing the data in the file, if you simply want to give the file a new name.

Example 4-11: Renaming a Scratched Data File Which Is Still Viable

```
10 SCRATCH "DATFIL-1"
20 DATASAVE DC OPEN F ("DATFIL-1") "TEST2"
```

Statement 10 scratches DATFIL-1. Statement 20 renames DATFIL-1 with the name "TEST2". The data in the file is not disturbed. However, the end-of-file trailer record in the file is lost and the USED column for TEST2 in the Catalog Index is reset to 1. Thus, you should note the sector address of the trailer record in DATFIL 1 prior to scratching it. After opening TEST2, you can skip to that location and rewrite the end-of-file record. Throughout this operation, the data is unaltered.

NOTE:

Although the name of a scratched file no longer appears in the catalog listing once the file has been renamed, the entry for the scratched file name remains in the Catalog Index. Thus, if a single file is scratched and renamed 16 times, only the final name appears in the catalog listing, despite the fact that all 16 names remain in the Catalog Index itself. Those 16 names would occupy one entire sector of the Catalog Index. Scratched file names can be removed from the Index only by executing a MOVE. The single exception to this rule is the case in which a scratched file is renamed with the same name. In that case, the same slot on the Catalog Index is used, and no duplication occurs. If it is necessary to scratch and rename files frequently, therefore, provision must be made for the scratched file names when establishing the size of the Catalog Index initially with SCRATCH DISK. Remember that the size of the Index cannot be altered once the catalog has been created.

4.6 EFFICIENT USE OF DISK STORAGE SPACE

The large storage capability of the disk unit may occasionally tempt the programmer to become inefficient in his use of disk storage space. Specifically, he may be tempted to design his records without due care for packing a maximum amount of data in a minimum number of sectors. Even when the available storage clearly exceeds present needs, however, this temptation should be overcome. Files have a way of outgrowing preliminary estimates at a faster-than-expected rate. Also, a file which is compact can be searched more quickly than one which is loosely layed out and contains large amounts of wasted space. In order to organize data within a record efficiently, it is necessary to understand more precisely how the system stores data in a sector. There are two main points to be considered:

1. Control information: The system automatically records control information along with the data in each sector. The control information occupies space in the data field of a sector, and must be taken into account when calculating how much space is required for a given amount of data.

2. "Gaps" in multisector records: Under certain conditions, gaps may occur between fields in a multisector record. In order to optimize the use of disk storage space, such gaps must be kept to a minimum.

System Control Information

The System automatically writes control information in each record created with a DATASAVE DC statement (or DATASAVE DA statement). This information is of two types:

1. Sector control bytes.
2. Start-of-value (SOV) control bytes.

Three sector control bytes are automatically written in each sector of a logical data record. The first two sector control bytes occupy the first two locations in the sector. The third control byte follows the last byte in the last field in the sector, and marks the end of valid data within that sector. Information in the sector following the last sector control byte (also called the "end-of-block" byte) is regarded as garbage, and is ignored by the system when the sector is read. After taking into account the three sector control bytes, only 253 of the 256 bytes in a sector are initially available for data storage.

In addition to the sector control bytes, a start-of-value (SOV) control byte is prefixed to every field stored in the sector. The SOV byte separates data fields within a sector, marking the beginning of each individual value in the sector.

Consider, for example, the following statements:

```
10 DIM A$(2) 30
20 DATASAVE DC A$( ), B$, "ABCD", 123, N
```

The argument list in statement 20 contains six separate arguments, each of which is prefixed with an SOV control byte when saved on disk. (Remember that each element of an array constitutes a single argument. Since A$( ) has two elements, it must be counted as two arguments.) The logical record created by statement 20, therefore looks like this:

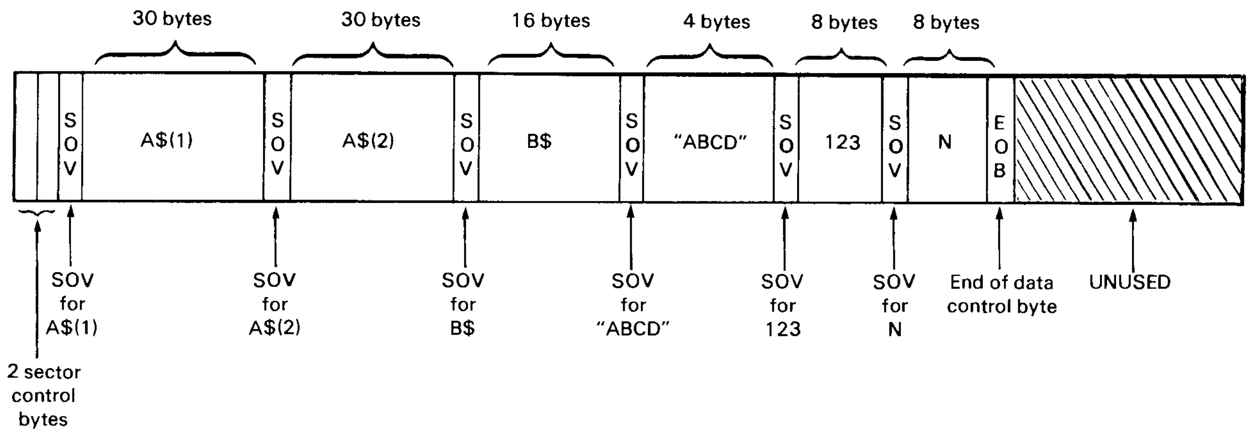


Figure 4-3. One Logical Record, Showing Sector Control Bytes and Start-of-Value Control Bytes for Each Field

From this illustration, the following disk storage requirements can be inferred:

a) Each numeric value, variable, or array element in the argument list always occupies nine bytes on disk (eight bytes for the numeric value and one byte for the SOV).

b) Each literal string in quotes occupies a number of bytes on disk equal to the number of characters in the literal string, plus one SOV byte.

c) Each alphanumeric variable or array element occupies a number of bytes on disk equal to the dimensioned length of the variable or element, plus one SOV byte.

Note that in the case of an alpha variable or array element, it is the dimensioned size, and not the number of characters actually stored in the variable or element, which must be counted. For example, the routine

```
50 DIM A$ 20
60 A$ = "ABC"
```

produces an alpha variable A$ which occupies 21 bytes on the disk (20 + 1), despite the fact that A$ contains a literal string only three characters in length. The remaining 17 bytes of A$ are blanks (spaces).

Inter-Field Gaps

In no case will the system overlap a single field from one sector to the next. If a field does not fit completely into one sector, it is written in its entirety into the next sequential sector. If record layouts are not carefully designed, this situation often gives rise to gaps between fields in multisector records.

Suppose, for example, that a logical record has been created with the following routine:

```
10  DIM A$(5)50, B$(3)64, C$48
 :
 :
100 DATASAVE DC A$(), B$(), C$
```

You could do some quick calculating, and making sure to add a control byte for each argument, conclude that the total record occupies 499 bytes. Since each sector can hold 253 bytes of data and control information (after the three sector control bytes are subtracted, two sectors can contain a total of 506 bytes). You might assume, therefore, that the record will fit easily into two sectors. Unfortunately, this calculation does not take into account the possibility of an inter-field gap. The argument list from line 100 is saved on disk in the following way:

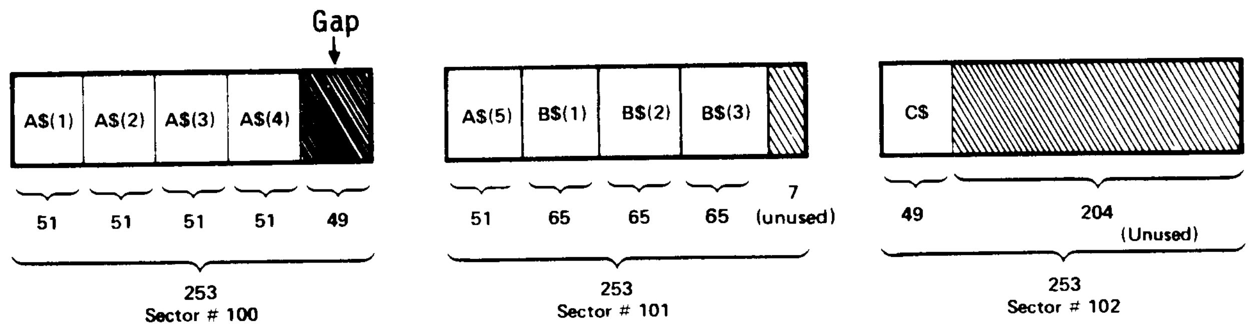


Figure 4-4. Inter-Field Gap in a Multi-Sector Record

Notice that the last field in sector 100 consists of 49 bytes, and is marked "unused". Since A$(5) requires 51 bytes of space, it does not fit into the remaining 49 bytes in sector 100, and the entire field is written into the next sector (sector #101). The unused 49 bytes in sector #100 represent a "gap" of wasted space between A$(4) and A$(5). As a result of this gap, C$ must be written in a third sector. Instead of requiring two sectors, as the figures indicated, this record occupies three sectors. If the file contains, say, 100 such records, it will require 100 more sectors than were initially estimated.

The waste resulting from inter-field gaps can, in many cases, be decreased or eliminated by careful attention to the design of the record. In this case, for example, the record can be made to fit into two sectors simply by rearranging the order of the arguments in the DATASAVE DC argument list:

```
100 DATASAVE DC C$, A$( ), B$( )
```

The resulting logical record now looks like this:

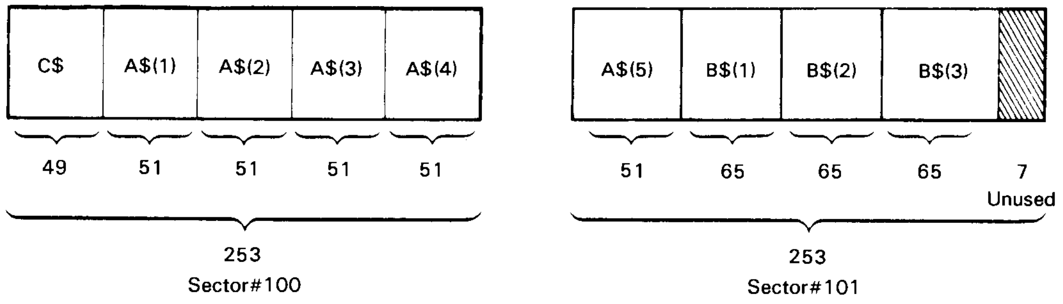


Figure 4-5. A Multi-Sector Record with No Gaps

By moving C$ from the end of the argument list to the beginning, the 49-byte gap in sector 100 is filled, thereby eliminating the need for a third sector in the record.

4.7 THE "LIMITS" STATEMENT

A special catalog statement, LIMITS, enables the programmer to obtain the sector address parameters of a cataloged file under program control. For catalog operations alone, LIMITS is useful in such ways as, for example, keeping track of the amount of free space remaining in a file during an input routine. When the catalog procedures are supplemented with Absolute Sector Addressing operations (discussed in Chapter 6), which provide direct access to individual sectors, LIMITS becomes a truly powerful programming tool. One important use of LIMITS in conjunction with Absolute Sector Addressing statements is in the binary search technique described in Chapter 6.

The LIMITS statement has two forms. In Form 1, the name of a cataloged disk file is specified in the LIMITS statement. In this case, LIMITS goes directly to the disk and retrieves the starting sector address, ending sector address, and number of sectors used for the named file from the Catalog Index. In Form 2, the file name is omitted from the LIMITS statement. When this form is used, LIMITS reads the sector address parameters from a specified slot in the Device Table (the default slot if no file number is specified), and retrieves the starting, ending, and current sector address parameters from that slot. In this case, the disk is never accessed.

Form 1 of LIMITS

In Form 1 of the LIMITS statement, the following information must be specified:

1. The disk platter on which the named file resides ('F','R',or 'T').
2. Optionally, a file number (#0-#15).
3. The name of the file whose parameters are to be obtained.
4. Three numeric return variables designated to receive the file parameters. Variable #1 is set equal to the starting sector address of the file, variable #2 is set equal to the ending sector address, and variable #3 is set equal to the number of sectors used in the file.
5. Optionally, a variable #4 designated to receive a value indicating the status of the specified file.

Form 1 of the LIMITS statement reads the Catalog Index entry for the named file and extracts the starting and ending addresses, and number of sectors used, and copies them into the three designated return variables (variables #1, #2, and #3). The status parameter variable #1 receives one of the following values indicating the status of the file:

Value	**Status**
2	active data file
1	active program file
0	filename not in index (in this case, variable #1 = variable #2 = variable #3 = 0)
-1	scratched program file
-2	scratched data file

Example 4-12: Form 1 of the LIMITS Statement ('File Name' Specified)

```
60 LIMITS F "TEST", A,B,C,D
```

Line 60 instructs the system to search the Catalog Index on the 'F' platter for the file "TEST", and retrieve the beginning and ending sector addresses of TEST, as well as the number of sectors used. These values are transferred to the variables A,B,C, and D according to the following scheme:

A = Starting sector address.
B = Ending sector address.
C = Number of sectors used.
D = 1 (if "TEST" is an active program file).

Example 4-13: Form 1 of the LIMITS Statement ('File Name' and a File Number Specified)

```
100 LIMITS T #2, "FILE-1", N,O,P,Q
```

Line 100 instructs the system to retrieve the file parameters of FILE-1 from the platter address listed under file #2 in the Device Table. The parameters are stored in the designated return variables N,O,P. If "FILE-1" is a data file, Q assumes the value 2.

Example 4-14: Form 1 of the LIMITS Statement ('File Name' Not in Catalog Index)

```
200 LIMITS R "DATA-2", T,O,N,Y
```

Line 200 instructs the system to retrieve the file parameters of DATA-2 from the 'R' platter. However, DATA-2 is not in the 'R' platter; therefore the sector variables are all set to zero as is the status variable, variable #4.

T = O = N = Y = zero.

Example 4-15: Form 1 of the LIMITS Statement ('File Name' Scratched)

```
300 LIMITS F #10, "CRAPS44", L,U,C,K
```

Line 300 instructs the system to retrieve the file parameters of CRAPS44 from the 'F' platter. The parameters are stored in the designated return variables L,U,C. The variable K takes on the value -2 to signify that CRAPS44 is a scratched data file.

Form 2 of Limits

In Form 2 of the LIMITS statement, the following information must be specified:

1. The 'T' parameter.
2. The file number (#0-#15) of a currently open file (if no file number is specified, the default file number, #0, is used).
3. Three numeric return variables designated to receive the file parameters. Variable #1 is set equal to the starting sector address of the file, variable #2 is set equal to the ending sector address, and variable #3 is set equal to the current sector address.

Form 2 of the LIMITS statement reads the sector address parameters from a specified slot in the Device Table, and stores them in the designated return variables. Unlike Form 1, Form 2 does not access the disk to read the Catalog Index.

Example 4-16: Form 2 of the LIMITS Statement ('File Name' Not Specified)

```
150 LIMITS T A,B,C
```

Line 150 reads the sector address parameters (starting, ending, current) from the default slot in the Device Table (since no file number is specified), and stores them in variables A,B,C in the following order:

A = Starting sector address.
B = Ending sector address.
C = Current sector address.

Example 4-17: Form 2 of the LIMITS Statement ('File Name' Not Specified)

```
200 LIMITS T #3, N,O,P
```

Line 200 retrieves the sector address parameters from the Device Table slot opposite file number #3, and stores those parameters in variables N,O,P.

Note that Form 2 of the LIMITS statement makes no check on the validity of the information read from the Device Table. It is the programmer's responsibility to ensure that the specified file number is associated with a currently open cataloged file. Because Absolute Sector Addressing operations do not store meaningful file parameter information in the Device Table, LIMITS should not be used with files maintained in Absolute Sector Addressing Mode. (LIMITS may be used in conjunction with Absolute Sector Addressing procedures to process a cataloged file, however; see Section 6.7.)

4.8 CONCLUSION

The discussion of catalog procedures is now concluded. All of the characteristics of the several catalog statements and commands and their applications have been touched upon. The programmer who wishes to make the most efficient use of the catalog procedures should press on, however, and read Chapter 6, which deals with the Absolute Sector Addressing Mode. Absolute Sector Addressing statements and procedures can be used in conjunction with cataloging procedures to produce a more versatile and efficient disk management system. In particular, Chapter 6 discusses the "binary search" technique for directly accessing records in a cataloged file.

CHAPTER 5
AUTOMATIC FILE CATALOGING STATEMENTS AND COMMANDS

5.1 INTRODUCTION

This chapter contains capsule descriptions and general forms for the following Automatic File Cataloging statements and commands, listed alphabetically for ease of reference:

DATALOAD DC	LOAD (Command)
DATALOAD DC OPEN	LOAD (Statement)
DATASAVE DC	LOAD RUN
DATASAVE DC CLOSE	MOVE
DATASAVE DC OPEN	MOVE END
DBACKSPACE	SAVE
DSKIP	SCRATCH
LIMITS	SCRATCH DISK
LIST DC	VERIFY
	$FORMAT DISK

5.2 SYSTEM 2200VP/MVP DISK STATEMENTS AND COMMANDS

The distinction between a statement and a command requires some explanation. Statements are programmable instructions used to write programs in BASIC-2. Every line in a BASIC-2 program consists of one or more statements, each of which directs the system to perform a specific operation or sequence of operations. Commands are used by the operator to control system operations directly from the keyboard, and generally are not programmable. Commands are entered and executed immediately by the operator; they are not stored in memory as part of a program.

Almost all BASIC-2 instructions governing disk operations are statements, and, as such, may be executed either in Program Mode (i.e., on a numbered program line) or in Immediate Mode. The sequence of operations associated with a disk statement when it is executed within a program is identical to the sequence of operations associated with the statement when it is executed in Immediate Mode.

A single exception to this rule, however, is represented by LOAD (and LOAD DA). The sequence of operations initiated by a LOAD (or LOAD DA) instruction when it is executed in Immediate Mode is significantly different from the sequence of operations initiated by the same instruction when executed on a numbered program line. For this reason, the LOAD instruction is treated as two separate and distinct entities, distinguished by their mode of execution: the LOAD statement (executed in a program), and the LOAD command (executed in Immediate Mode). LOAD DA is treated similarly in Chapter 7.

The only command in the BASIC-2 disk instructions is LOAD RUN. This command cannot be executed in a program.

5.3 BASIC RULES OF SYNTAX AND TERMINOLOGY

Rules of Syntax

The notation and rules of syntax employed in the General Forms of disk statements follow the conventions used in the System 2200VP/MVP BASIC-2 Language Reference Manual. The conventions are summarized below:

1. The following symbols must be included in an actual BASIC statement exactly as they appear in the General Form of the statement:

a.	Uppercase letters	A through Z
b.	Comma	,
c.	Double Quotation Marks	"
d.	Parentheses	( )
e.	Pound Sign	#
f.	Slash	/

2. Lowercase letters and words in the General Form of a statement represent items whose values must be assigned by the programmer. For example, if the lowercase word "file-name" appears in a General Form, the programmer must substitute a specific file name (such as "PROG 1"), or an alphanumeric variable containing the name, in the actual statement.

3. Three special symbols are used in the General Forms to indicate optional, alternate, or repetitive items. These symbols are never included in an actual BASIC statement:

a.	Brackets	[ ]
b.	Braces	{ }
c.	Ellipses	...

4. Square brackets, [ ], indicate that the enclosed information is optional, and may be included or not in the actual BASIC statement, at the programmer's discretion.

5. Vertically stacked items represent alternatives, only one of which should be included in an actual BASIC statement:

   a. Square brackets, [ ], enclosing vertically stacked items, indicate that all of the items are optional.

   b. Braces, { }, enclosing vertically stacked items, indicate that one of the items must be included in an actual statement.

6. Ellipses, ..., indicate that the preceding item(s) may be repeated once or several times in succession.

7. Blanks (spaces); used to improve the readability of the General Forms, are meaningless to the system (unless enclosed in double quotation marks), and may be omitted or included in an actual statement, at the option of the programmer.

8. The sequence in which terms are listed in the General Form of a statement must be followed exactly in an actual statement.

Definition of Terms Used in General Forms

The following terms are used frequently in the General Forms of disk statements and commands in this chapter and in Chapter 7. They are defined here rather than repeated for each statement or command.

BA = a parameter specifying Absolute Sector Address Mode and block data format.

DA = a parameter specifying Absolute Sector Address Mode and standard System 2200 data format.

DC = a parameter specifying Disk Catalog Mode.

disk address = /xyy

where: x = device type (see Section 1.4).

yy = unit device address of disk.

If neither a file # nor a disk-address is explicitly specified in a disk statement, the default Device table slot (#0)is used.

file name = $\left\{\begin{matrix}\text{literal-string}\\ \text{alpha-variable}\end{matrix}\right\}$, length of file name must be from 1 to 8 characters. The file-name specifies a cataloged disk file.

file# = #$\left\{\begin{matrix}\text{integer}\\ \text{numeric-variable}\end{matrix}\right\}$, the value of the integer or value of the variable must be from 0 to 15. The file # specifies the Device Table Slot to be used by the disk statement. The Device Table Slot contains the disk address (assigned by a SELECT statement) and file 'start', 'end', and 'current' sector address information for certain Disk Catalog Mode statements. If neither a file # nor a disk address is explicitly specified in a disk statement, the default Device Table slot (#0) is used.

platter = $\left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\}$

where: F specifies the fixed platter of a Model 2260 series drive, the platter in the first drive of a 2270/2270A, the diskette in the first drive of a minidiskette, or the diskette in the third drive of a 2270-3/2270A-3.

R specifies the removable platter of a Model 2260 series drive, the diskette in the second drive of a 2270/2270A or the diskette in the second drive of a minidiskette.

T specifies either 'F' or 'R' depending on the device type specified in the disk address.

On the Model 2260 and 2270 Series, device type 3 implies 'F'; device type B implies 'R'.

Platter specification T must be used whenever device type 'D' is used to access a disk.

$ = a parameter specifying read-after-write verification is to be performed on all data written to disk. Read-after-write not only detects improperly written information, but also effectively doubles the time required for the write operation.

General Form:

DATALOAD DC [file#,] argument-list

where:

$$\text{argument-list} = \left\{\begin{matrix}\text{variable}\\ \text{array}\end{matrix}\right\} \left[, \left\{\begin{matrix}\text{variable}\\ \text{array}\end{matrix}\right\} \right] \ldots$$

Purpose:

The DATALOAD DC statement is used to read logical data records from a cataloged disk file and sequentially assign the values read to the variables and/or arrays in the argument list. An error results if numeric data is assigned to an alpha variable, or vice versa. If data assigned to an alpha variable is shorter than the length of the variable, the value is padded with trailing spaces; if the value is longer, it is truncated. Before data can be read from a cataloged file, the file must be opened by a DATALOAD DC OPEN or DATASAVE DC OPEN statement. Thereafter, each time a DATALOAD DC statement is executed, the system begins reading data from the file at the next sequential logical record in the file. Arrays are filled row by row. Notice that alpha arrays (e.g., A$()) receive a separate data value for each element of the array. However, the STR () of an array receives only one value.

If the DATALOAD DC receiving variable list is not filled by one logical record, the next logical record is read. If the logical record being read contains more data than is required to fill all receiving variables in the argument list, data not used is read but ignored. Each time the DATALOAD DC statement is executed, the Current Sector Address associated with the file in the Device Table is updated to the Starting Sector Address of the next consecutive logical record. If an end-of-file trailer record is read, an end-of-file condition is set, the Current Sector Address is set to the address of the trailer record, and no data is transferred. The end-of-file condition can be tested by a subsequent IF END THEN statement. If the user attempts to read beyond the final sector address for the file, an error is signalled.

Examples of valid syntax:

```
100 DATALOAD DC S(), Y, Z
100 DATALOAD DC #2, A$(), B()
100 DATALOAD DC #B2, B(), C, D$
```

DATALOAD DC OPEN

General Form:

```
                                       ⎧file-name                           ⎫
DATALOAD DC OPEN   platter [file #,] ⎨                                      ⎬
                                       ⎩TEMP[ , ]start-sector, end-sector   ⎭
```

where:

TEMP	= A temporary work file is to be reopened.
start-sector	= An expression whose value is the starting sector address of the temporary work file.
end-sector	= An expression whose value is the ending sector address of the temporary work file.

Purpose:

The DATALOAD DC OPEN statement is used to open data files that have previously been cataloged on the disk. When the statement is executed, it locates the named file on the specified disk platter, and sets up the starting, current, and ending sector addresses of the file in the Device Table (the current address is set equal to the starting address). The platter digit in the file status column of the Device Table reflects which platter the file was open on ('1' if 'F' and '2' if 'R'). Any subsequent use of the same file number in other catalog (DC) statements accesses this file. If no file number is included, the file is assumed to be associated with the default file number (#0) and can be accessed by subsequent DC statements with the file number omitted, or by specifying file # = #0.

An error will result if the file name cannot be located in the Catalog Index of the specified disk, or if the file has been scratched.

The TEMP parameter is used to re-open a temporary work file; the starting and ending addresses must not be located in the cataloged area. Temporary file areas can be accessed with catalog statements and commands (e.g., DATASAVE DC, DATALOAD DC, etc.). Caution must be exercised when using temporary files in a multi-user or multiplexed disk environment. Here the temporary file start-sector and end-sector parameters are maintained local to each CPU. The system does not restrict two or more users from requesting the same temporary file space on disk.

The DATALOAD DC OPEN statement must be used when reopening an existing cataloged data file; use of the DATASAVE DC OPEN statement results in an error if the named file is already in the catalog and has not been scratched. Therefore, DATALOAD DC OPEN is used to reopen a cataloged file irrespective of whether data is to be written in the file with a DATASAVE DC statement or read from the file with a DATALOAD DC statement.

Examples of valid syntax:

```
100 DATALOAD DC OPEN F "HEADING"
100 DATALOAD DC OPEN R #2, A$
100 DATALOAD DC OPEN T #A, TEMP 8000, 9000
100 DATALOAD DC OPEN T "PARTFIL"
```

General Form:

$$\text{DATA SAVE DC [\$] [file \#,]} \begin{Bmatrix} \text{END} \\ \text{argument-list} \end{Bmatrix}$$

where:

$$\text{argument-list} = \begin{Bmatrix} \text{literal} \\ \text{variable} \\ \text{expression} \\ \text{array} \end{Bmatrix} \left[, \begin{Bmatrix} \text{literal} \\ \text{variable} \\ \text{expression} \\ \text{array} \end{Bmatrix} \right] \cdots$$

END = Write a data trailer (end-of-file) record.

Purpose:

The DATASAVE DC statement causes one logical record, consisting of all the data in the DATASAVE DC argument list, to be written onto the disk, starting at the current sector address associated with the specified file number (#n) in the Device Table. If no file number is specified in the DATASAVE DC statement, the data is written into the file currently associated with the default file number (#0) in the Device Table. The file must previously have been opened with a DATASAVE DC OPEN or DATALOAD DC OPEN statement. No data can be saved into an unopened file; if the DATASAVE DC statement specifies a file number not associated with a currently open file, an error results.

The DATASAVE DC argument list may include literal strings (e.g., "JOHN JONES") and expressions (e.g., B*C), as well as alphanumeric and numeric variables and arrays.

The 'DC' parameter implies that the data in the argument list is to be written as one logical record in standard System 2200 format, including the necessary control information. The values in the argument list are stored sequentially on the specified disk. Arrays are written row by row. It should be noted that each element of an array is written as a separate data value. However, the STR ( ) of an alpha array represents a single data value. Alphanumeric values must be $\leq$ 124 bytes in length. Each single logical record may consist of one or more sectors on the disk.

NOTE:

Each numeric value in the argument list requires 9 bytes of storage on disk. Each alphanumeric variable requires the maximum length to which the variable is dimensioned plus 1 byte; e.g., if the length of A$ is set to 24 characters in a DIM A$24 statement, then A$ requires 25 (24 + 1) bytes of storage on disk. Each 256 byte sector also requires 3 bytes of sector control information (refer to Section 7.6).

The '$' parameter specifies that a read-after-write verification test be made on all data written to the disk. This test not only provides an extra safeguard against disk write errors, but it also substantially increases the time required for the DATASAVE DC operation.

If the END parameter is specified, a data trailer record is written in the file, and the Catalog Index entry for the file is updated so that the number of sectors used by the file includes all sectors up to the trailer record just written. A cataloged file always should be ended by a trailer record. A new data record can be stored in the file by writing over the trailer record, and subsequently creating a new trailer record. (A DSKIP END statement positions the system to the beginning of the trailer record; a DATASAVE DC statement can be executed at that point to store the new record over the trailer record, and a subsequent DATASAVE DC END statement executed to create a new trailer record.)

Examples of valid syntax:

```
100 DATASAVE DC A,X, "CODE#4"
100 DATASAVE DC $ #2, M$, P2( ), F1$( )
100 DATASAVE DC $ #1, "ADDRESS", (3*1)/100, J$( )
100 DATASAVE DC #3, END
100 DATASAVE DC #A, A$( )
```

General Form:

```
                          ┌ file # ┐
DATA SAVE DC CLOSE        │        │
                          └ ALL    ┘
```

where:

ALL = All currently open files are to be closed

Purpose:

The DATASAVE DC CLOSE statement is used to close an individual data file or all data files which are currently open, if they are no longer needed in the current or subsequent programs. The DATASAVE DC CLOSE statement closes a file by setting the starting, ending, and current sector addresses and platter digit associated with its file number in the Device Table equal to zero. When the file is closed, a disk statement referencing that file causes an ERROR D80 (File Not Open) to be displayed.

If the file # parameter is used, the single file associated with that file number is closed. If the ALL parameter is used, every open file is closed. If neither parameter is used, the currently open file associated with the default file number (#0) is closed.

The DATASAVE DC CLOSE statement should not be confused with DATASAVE DC END. The latter writes an end-of-file record at the end of a newly written file. The end-of-file record should always be written prior to executing DATASAVE DC CLOSE.

It is good programming practice to close a file with DATASAVE DC CLOSE upon completion of processing, since it ensures that subsequent disk users will not erroneously access the file and possibly destroy data. Likewise, DATASAVE DC CLOSE can be used at the beginning of a program to initialize file parameters to zero before they are set by DATASAVE DC OPEN or DATALOAD DC OPEN. DATASAVE DC CLOSE does not remove disk device addresses from the Device Table.

Examples of valid syntax:

```
900 DATASAVE DC CLOSE
900 DATASAVE DC CLOSE #3
900 DATASAVE DC CLOSE ALL
900 DATASAVE DC CLOSE #A
```

DATASAVE DC OPEN

General Form:

```
                                            ⎧ ⎛old-file-name⎞  new-file name ⎫
DATASAVE DC OPEN platter [ $ ] [ file #, ] ⎨ ⎝space        ⎠                ⎬
                                            ⎩ TEMP[ , ] start-sector, end-sector ⎭
```

where:

old-file-name	= The name of an existing scratched program or data file which is cataloged on the specified disk platter.
space	= An expression signifying the number of sectors to be reserved for a new file.
new-file-name	= The name of the data file being opened.
TEMP	= A temporary work file is to be established.
start-sector	= An expression whose value is the starting sector address of a temporary work file.
end-sector	= An expression whose value is the ending sector address of a temporary work file.

Purpose:

The DATASAVE DC OPEN statement is used to reserve space for cataloged files in the Catalog Area, and to enter appropriate system information in the Catalog Index. It is also used to reserve space for temporary work files outside the Catalog Area, and to reuse space in the Catalog Area occupied by scratched files.

Data files can be opened on any disk platter by including the proper parameter ('F' or 'R') in the DATASAVE DC OPEN statement. Each data file must be opened initially with a separate DATASAVE DC OPEN statement; if multiple files are to be opened simultaneously, each file must be assigned a different file number. Since there are 16 file numbers available (0-15), a total of 16 data files can be opened simultaneously.

The '$' parameter specifies that a 'read-after-write' verification test be performed to ensure that the file and all file control information is written correctly in the Catalog Index. This test not only detects disk write errors, but also substantially increases the time required for the DATASAVE DC OPEN operation.

The 'file #' parameter is the file number which identifies the newly-opened file in the Device Table. The disk on which the file is stored, along with the file's starting, ending, and current sector addresses, are entered in the Device Table in System 2200VP/MVP memory. Also, the platter digit under the 'file status' column of the Device Table reflects which platter the file was opened on ('1' for 'F', and '2' for 'R'). The information in the Device Table is identified only by the file number assigned to the file in the DATASAVE DC OPEN statement. A file number must be included in the DATASAVE DC OPEN statement if more than one file is to be open at one time. If no file number is specified, or if file # = #0, the system automatically assigns the newly opened file to the default slot, #0, in the Device Table. Subsequent reference to a file number in a disk catalog statement or command automatically provides access to the current sector address of the associated file. (For a detailed discussion of the Device Table and the use of file numbers, see Chapter 6.)

The 'old-file-name' parameter specifies the name of a previously scratched cataloged file (either program or data) which is to be renamed and reused. The new file is given the space previously occupied by the scratched file.

If the 'space' parameter is used instead of 'old-file-name', the new file is appended at the current end of the Catalog Area, and given a total number of sectors equal to the value of the 'space' expression.

NOTE:

The last sector of each cataloged data file is reserved for systems information. Therefore, the number of sectors available for data storage is always at least one less than the number of sectors reserved for the file.

The 'new-file-name' parameter is the name of the new data file being opened. If the new file is being stored in space previously occupied by a scratched cataloged file ('old'), then 'new' can be identical to 'old'. Otherwise, 'new' must be unique.

The TEMP parameter is used to specify a temporary work file. Temporary files are not cataloged and cannot be located in the Catalog Area. If temporary files are to be used, sufficient space must be left outside the Catalog Area to accommodate them (see SCRATCH DISK). Caution must be exercised when using temporary files in a multi-user or multiplexed disk environment. Here the temporary file start-sector and end-sector parameters are maintained local to each CPU. The system does not restrict two or more users from requesting the same temporary file space on disk.

The 'start' and 'end' parameters identify the starting and ending sectors of the area reserved for a temporary file. An error results if the value of 'start' is less than or equal to the last (highest) sector of the Catalog Area.

Examples of valid syntax:

```
100 DATASAVE DC OPEN R (100) "DATFIL1"
100 DATASAVE DC OPEN R #1, (A*2) "I/O DATA"
100 DATASAVE DC OPEN F #2, ("DATFIL1") "DATFIL2"
100 DATASAVE DC OPEN F TEMP 1000, 2000
100 DATASAVE DC OPEN T$#4, (200) A$
```

DBACKSPACE

General Form:

```
                            ⎧ BEG            ⎫
DBACKSPACE [ file #, ]  ⎨                ⎬
                            ⎩ expression[ S ] ⎭
```

where:

BEG	= Backspace to beginning of file.
expression	= The number of logical records or sectors to be backspaced.
S	= Backspace absolute number of sectors.

Purpose:

The DBACKSPACE statement is used to backspace over logical records or sectors within a cataloged disk file. If 'expression' is used alone, the system backspaces over the number of logical records equal to the value of the 'expression', and the Current Sector Address of the file in the Device Table is updated to the starting sector of the new logical record. For example, if 'expression' = 1, the Current Sector Address is set equal to the starting address of the previous logical record. If the BEG parameter is used, the Current Sector Address is set equal to the Starting Sector Address of the file (that is, the starting address of the first logical record in the file).

If the 'S' parameter is used, the value of the expression equals the total number of sectors to backspace. The Current Sector Address of the file in the Device Table is decremented by the number of sectors specified. If the amount specified is too large, the Current Sector Address is set to the starting Sector Address of the file. The 'S' parameter is particularly useful in files where all the logical records are of the same length (i.e., have the same number of sectors per logical record). Backspacing with the 'S' parameter is much faster than backspacing over logical records in a file since the system merely decrements the Current Sector Address in the Device Table by the specified number of sectors and no disk accesses are required. However, the user must be certain that he knows exactly how many sectors are in each logical record.

Examples of valid syntax:

```
100 DBACKSPACE BEG
100 DBACKSPACE 2*X
100 DBACKSPACE #2, 5S
100 DBACKSPACE #1, BEG
100 DBACKSPACE #A, 10
```

General Form:

$$\text{DSKIP [file \#,]}\begin{Bmatrix}\text{END}\\ \text{expression [S]}\end{Bmatrix}$$

where:

END	= skip to current end-of-file.
expression	= The number of logical records or sectors to be skipped.
S	= Absolute number of sectors are to be skipped.

Purpose:

The DSKIP statement is used to skip over logical records or sectors in a cataloged disk file. If 'expression' is used alone, the system skips over a number of logical records equal to the value of 'expression', and the Current Sector Address for the file is updated to the starting address of the new logical record. If the 'END' parameter is used, the system skips to the end of the file, i.e., the current sector address for the file is updated to the address of the end-of-file trailer record. Once a DSKIP END statement has been executed, data can be added to the end of the file using DATASAVE DC statements. Note that the DSKIP END statement cannot be used unless a trailer record has previously been written in the file with a DATASAVE DC END statement. DSKIP END results in an Error D87 (No End of File) if no trailer record can be located in the file.

If the 'S' parameter is used, the value of the expression equals the total number of sectors to be skipped. The Current Sector Address of the file is incremented by the number of sectors specified. If the amount specified is too large, the Current Sector Address is set to the Ending Sector Address of the file. The 'S' parameter is particularly useful in files where all logical records are of the same length (i.e., have the same number of sectors per logical record). Skipping with the 'S' parameter is much faster than skipping logical records in a file since the system merely increments the current address by the specified number of sectors and no disk accesses are necessary. However, the user must be sure that he knows exactly how many sectors are in each logical record.

Examples of valid syntax:

```
100 DSKIP 4
100 DSKIP #2, END
100 DSKIP END
100 DSKIP #3, 4*X
100 DSKIP #A, 20S
```

LIMITS

General Form:

Form 1:

LIMITS platter [file #,] file-name, start, end, used [,status]

Form 2:

LIMITS platter [file #,] start, end, current

where:

file name	= The name of the cataloged data or program file whose limits are to be retrieved (Form 1). If 'file-name' is not specified (Form 2), limit information on a currently open file (in the Device Table) is to be retrieved.
start	= A numeric variable designated to receive the starting sector address of the file.
end	= A numeric variable designated to receive the ending sector address of the file.
used	= A numeric variable designated to receive the number of sectors used by the file.
current	= A numeric variable designated to receive the current sector address of the file.
status	= A numeric variable designated to receive a value indicating the status of the specified file.

Purpose:

The LIMITS statement obtains the beginning and ending sector address and current sector address or number of sectors used for a cataloged file. In addition, LIMITS determines the status of the specified file.

LIMITS can be used within a program to find out the status of a file, how much remaining space is left in a file, or to get sector address limits of an active file.

Form 1: Limits of a Cataloged File ('file-name' Specified)

If a file-name is specified, the LIMITS statement finds the named program or data file on the specified disk and sets the 'start' variable equal to the starting sector address of the file, the 'end' variable equal to the ending sector address of the file, the 'used' variable equal to the number of sectors currently used by the file, and the 'status' variable equal to a value that specifies the status of the file. The number of sectors currently being used by the file is accurate only if an end-of-file record has been written in the file. An end-of-file record is written in a data file with a DATASAVE DC END statement.

Therefore, in order to be able to tell how many sectors are used in a data file, the file must be ended with an end-of-file record.

If a file-name is specified and a 'status' variable is also specified, the LIMITS statement causes the 'status' variable to receive a value indicating the status of the file. The following values may be assigned to the variable.

```
                  {2 if active data file
                  {1 if active program file
'status' variable= {0 if file name not in index (in this case
                  { variables 'start', 'end', and 'used'
                     will also be set to zero)
                  {-1 if scratched program file
                  {-2 if scratched data file
```

Examples of valid syntax:

```
100 LIMITS F "PAYROLL", A,B,C,D
100 LIMITS T A$, S,E,A
100 LIMITS T #A, "DATFIL 1", X,Y,Z(3)
100 LIMITS F #1, "SAM", A,B,C
```

Note that in Form 1 of the LIMITS statement if the 'status' variable is not specified, an error will result if the specified file does not exist.

Example:

```
100 LIMITS F "PRICE",A,B,C (and "PRICE" is on the "R" platter)

                                    ↑ ERR D82
```

Form 2: Limits of a Currently Open File ('file-name' Not Specified)

If a file name is not specified, the LIMITS statement gives the starting, ending, and current sector addresses of the file currently open at file # or in the default slot. The disk is not accessed; the data is read directly from the Device Table.

Examples of valid syntax:

```
100 LIMITS T #A(1), A1,A2,A3
100 LIMITS T #5, A,B,C
100 LIMITS T X,Y,Z(2)
```

LIST DC

General Form:

$$\text{LIST[S] [title] DC platter} \begin{bmatrix} \text{file \#} \\ \\ \text{disk-address} \end{bmatrix}$$

where:

S = A parameter indicating that the Disk Catalog Index is to be listed in sections.

$$\text{title} = \begin{Bmatrix} \text{literal-string} \\ \\ \text{alpha-variable} \end{Bmatrix}$$

Purpose:

The purpose of the LIST DC statement is to display or print out a listing of the information contained in the Catalog Index. When the List DC statement is executed, the following information is displayed on the currently selected LIST device:

a. The number of sectors in the Catalog Index.
b. The address of the last sector reserved for the Catalog Area.
c. The address of the last used sector in the Catalog Area.

For each cataloged file, the LIST DC statement outputs the following data:

a. The file name.
b. The file status (S if scratched).
c. The file type (program (P) or data (D)).
d. The Starting Sector Address.
e. The Ending Sector Address.
f. The number of sectors currently used in the file. For a data file, this value is originally set to one, and is updated only when an end-of-file record is written in the file.
g. The number of sectors not used in each file.

If the entry in the index is invalid, a '?' is displayed rather than the 'S', 'P' or 'D'.

The 'S' parameter is a special feature for the CRT terminal. It permits the listing of the catalog in sections, that is, listing stops when the screen is full. To continue listing, (EXEC) must be keyed. Note that for nonstandard CRT's, the number of lines on the CRT is specified by a SELECT LINE statement. The 'S' parameter is ignored in Program Mode.

Keying HALT/STEP during the listing stops the listing after the current line has been listed. However, the listing cannot be continued. Alternatively, the operator may slow down listing on the CRT by selecting a pause of from 1/6 to 1 1/2 seconds by executing a SELECT P statement. In this case, the system pauses for the specified interval after each line is listed.

The optional "title" parameter is a convenient means of identifying a hardcopy catalog listing. If a title is included in the LIST DC command, the system performs the following actions:

a. Issues a top-of-form (HEX(0C)) code.
b. Prints the title in expanded print.
c. Skips a line.
d. Prints the listing.

On the printer, these actions cause the title to be printed at the top of a new page in expanded print, followed by a blank line and the catalog listing. On a CRT, the title is displayed before the catalog listing.

When the system is Master Initialized, the CRT is initially selected for all LIST operations. Other printing devices may be selected for listing with a SELECT LIST statement.

Examples of valid syntax:

```
LIST DC F
LIST DC F #2
LISTS DC R
LIST DC T
LIST DC T #A
LIST S DC F/320
LIST "DISK#2" DC R
LIST "DISK#5" DC T/D15
LIST DC T/D10
```

LOAD (Command)

General Form:

$$\text{LOAD [DC] platter} \begin{bmatrix} \text{file \#,} \\ \text{disk-address,} \end{bmatrix} \text{file-name}$$

where:

file-name = The name of the cataloged program to be loaded.

Purpose:

The LOAD command is used to load BASIC programs or program segments from the disk. This command causes the system to locate the named program in the catalog, and append it to the program text currently in memory. Programs can be loaded into memory from any disk platter.

LOAD can be used to add to program text currently in memory, or if executed following a CLEAR command, to load a new program. An error results if the requested file is not a program file, or if it is not present in the catalog.

Examples of valid syntax:

```
LOAD F "PROG1"
LOAD R #2, "TESTI/O"
LOAD R /320, "OUTPUT1"
LOAD T A$
LOAD DC T #A1, B$
LOAD T/D11, "PARFILE"
LOAD T/B10, "DATFILE"
```

General Form:

```
                 [ file #,      ] ( file-name                      )
LOAD [DC] platter[              ] {                                } [line-no. -1], [line-no. -2]
                 [ disk-address ] ( <expression> alpha-variable    )             [BEG begin-line-no.]
```

where:

file-name	= The name of the cataloged program file to be loaded into memory.
expression	= The number of files to be loaded from disk.
alpha-variable	= The names of the files to be loaded. Names are 8 characters in length (padded with trailing spaces if necessary) and are stored sequentially in the alpha-variable. The alpha-variable must be a common variable.
line-number-1	= The number of the first program line to be deleted from the program currently in memory prior to loading the new program.
line-number-2	= The number of the last program line to be deleted from the program currently in memory before the new program is loaded.
begin-line-number	= The line number of the program where execution is to begin after the program is loaded into memory.

Purpose:

The LOAD statement loads a BASIC program or program segment into memory from the disk, and automatically executes it. LOAD is a BASIC statement which in effect produces an automatic combination of the following BASIC statements and commands:

STOP	(Stops current program execution and makes sure the file exists.)
CLEAR P	(Clears program text from memory, beginning at 'line 1' (if specified) and ending at 'line 2' (if specified); if no line numbers are specified, clears all currently stored program text from memory.) The Interrupt Table and subroutine stacks in memory are also cleared.
CLEAR N	(Clears all non-common variables from memory.)
LOAD DC	(Loads new program or program segment from disk.)
RUN	(Runs new program, beginning at 'begin-line-no', if specified. If the BEG parameter is not specified, program execution begins at line-number-1, or at the lowest program line in memory, if line-number-1 is not specified.)

If 'line 2' is omitted, the remainder of the currently stored program is deleted, starting with line-number-1, prior to loading the new program from disk. If line-number-1 is omitted, all program lines up to and including line-number-2 are deleted. If both line numbers are specified, all program lines in memory between and including these lines are cleared prior to loading the new program. If no line numbers are specified, all currently stored program text is cleared. In all cases, all non-common variables are cleared prior to loading the new program.

The LOAD statement permits segmented programs to be run automatically without normal user intervention. Common variables are passed between program segments. If LOAD is included on a multistatement line, it must be the last executable statement on the line.

The LOAD statement can be used to load several programs from disk when the <expression> parameter and an alpha-variable are specified. This feature allows a reduction in program overlay time (since the program is resolved only after all overlays have been loaded, rather than after loading each overlay), and provides more flexibility for program assembly from several program modules.

Examples of valid syntax:

```
100 LOAD R "PROG1"
100 LOAD F #2, "I/OMSTR"
100 LOAD DC F/320, "I/OSUB1" 250, 299
100 LOAD R "I/OCNTRL" 500
100 LOAD T A$ 100
100 LOAD DC T #X, B$
100 LOAD F "R. NIXON" 10, 500 BEG 300
100 LOAD T/D12, "INFILE"
```

Example of multiple file LOAD statement:

```
:10 COM A$(5)8
:20 A$(1) = PROG1": A$(2) = PROG2": A$(3) = "PROG3"
:30 LOAD F<3> A$()
```

Line 30 specifies that 3 program files (PROG1, PROG2, PROG3) are to be loaded, in that order.

NOTE:

When a Load statement is executed, the system stacks are cleared of all subroutine and loop information.

In Immediate Mode, LOAD is interpreted as a command (see LOAD command).

LOAD RUN (Command)

General Form:

```
                         ⎡ file #,        ⎤
LOAD RUN [ platter ]     ⎢                ⎥  [file-name ]
                         ⎣ disk-address,  ⎦
```

where:

T	= The default platter parameter if platter is not specified.
file-name	= The name of the cataloged program to be loaded. The default file-name is "START".

Purpose:

The LOAD RUN command is used to load and initiate execution of a program stored on disk. This command produces an automatic combination of the following operations:

1. CLEAR all program text and removes all variables from memory. PRINT, PRINTUSING, and LIST operations are selected to current Console Output device; INPUT, LINPUT, and KEYIN operations are selected to the current Console Input device. Device Table slots #1-#15 and the file parameters in #0 are set to zero. Pause (SELECT P) and TRACE are turned off. The Interrupt Table and subroutine stacks in memory are cleared. Radians are selected for trig functions and SELECT ERROR > 60 is established for computational error control. The ERR function is set to zero. (See CLEAR in BASIC-2 Reference Manual.)

2. LOAD the named program from the designated platter.

3. RUN the program.

Since default values are provided for the platter, disk-address, and filename parameters, a system initialization program named "START" can be loaded and run by pressing 3 keys on the system keyboard: 'LOAD' 'RUN', and 'RETURN (EXEC)'. The "START" program can then provide a menu and entry points of available programs.

Examples of valid syntax:

```
:LOAD RUN
:LOAD RUN R "BEGIN"
:LOAD RUN R /320, "NAMES"
:LOAD RUN R
:LOAD RUN T/D13, "CITIES"
```

MOVE

General Form:

Form 1:

$$\text{MOVE platter} \begin{bmatrix}\text{file\#,}\\ \\ \text{disk-address,}\end{bmatrix} \text{TO platter} \begin{bmatrix}\text{file\#,}\\ \\ \text{disk-address,}\end{bmatrix}$$

Form 2:

$$\text{MOVE platter} \begin{bmatrix}\text{file\#,}\\ \\ \text{disk-address,}\end{bmatrix} \text{file-name TO platter} \begin{bmatrix}\text{file\#,}\\ \\ \text{disk-address,}\end{bmatrix} \left[\left(\begin{bmatrix}\text{file-name}\\ \text{space}\end{bmatrix}\right)\right]$$

where:

space = An expression whose truncated value equals the number of sectors to be reserved in addition to the number required to store the program on the destination platter.

Purpose:

The purpose of the MOVE statement is to copy information from platter to platter within the same disk unit, or from one disk unit to another on the same system. Two forms of MOVE are provided:

Form 1 copies the entire Catalog.

Form 2 copies only a specified file.

Following a MOVE, the user can execute a VERIFY statement to ensure that the information was copied correctly.

When MOVE is executed, approximately 800 bytes of memory must be available for buffering (not occupied by a BASIC program or variables); otherwise an error A03 results and the MOVE is not performed. The large buffer minimizes the time required for the MOVE operation.

Form 1: Moving the Entire Disk Catalog

When the filename is not specified, execution of the MOVE statement copies the entire catalog of the origin platter, except for scratched files, to the destination platter. The starting, ending, current, and free sector addresses of all relocated files are automatically altered to reflect the files' new positions in the Catalog. Temporary files are not copied and the origin platter is not modified.

Examples of valid syntax:

```
10 MOVE R TO R/320,
10 MOVE R TO F
10 MOVE R/310, TO F/350,
10 MOVE T/D11, TO T/D10,
10 MOVE T/D12, TO T/D15,
```

Form 2: Moving a Specified File

When the filename is specified in the MOVE statement, the system causes the named file to be copied from the Catalog Area of the origin platter to the Catalog Area of the destination platter. The filename is entered in the Catalog Index on the destination platter. If the expression in parentheses is specified, the system reserves an additional amount of sectors for the named file in the Catalog Area of the destination platter. The truncated value of the expression indicates the number of additional sectors to be reserved.

If a destination filename is included within the parentheses, it is the scratched file to be overwritten by the new file when MOVE is executed.

If a scratched file with the same name as the file to be moved is to be overwritten, the destination filename can be omitted from the parentheses. For example,

```
SCRATCH R "DATAFILE"
MOVE F "DATAFILE" TO R ()
```

replaces the scratched file "DATAFILE" on the 'R' platter with the file called 'DATAFILE' from the 'F' platter.

Examples of valid syntax:

```
10 MOVE F "STOCKS" TO R
10 MOVE R/320, "PRICES" TO F/310,
10 MOVE R "DATA" TO F(100)
10 MOVE F "NAMES" TO R(2.5*V)
10 MOVE F "ADDRESS2" TO R("ADDRESS1")
10 MOVE F "DELTA" TO R()
10 MOVE T/D11, "BONDS" TO T/D10, ()
10 MOVE T/D15, "SHARES" TO T/D10,
```

MOVE END

General Form:

$$\text{MOVE END platter} \begin{bmatrix} \text{file \#} \\ \\ \text{disk-address} \end{bmatrix} = \text{expression}$$

Purpose:

The MOVE END statement is used to increase or decrease the size of the Catalog Area on a disk platter. The upper limit of the Catalog Area is initially defined by the END parameter in the SCRATCH DISK statement (see SCRATCH DISK). Once the limit of this area has been set, it can be altered using the MOVE END statement. The value of the 'expression' specifies the sector address of the new end of the Catalog Area. An error results if a previously cataloged file resides at this address, or if the address is higher than the highest legal address on the platter. Note that MOVE END does not alter the size of the Catalog Index.

Examples of valid syntax:

```
MOVE END F = 4799
MOVE END R = .5*L
MOVE END T#12 = X+Y
MOVE END R/320 = 2399
MOVE END T/D13 = 9400
```

General Form:

$$\text{SAVE [DC]} \begin{bmatrix} \text{< S >} \\ \\ \text{< SR >} \end{bmatrix} \text{platter [\$]} \begin{bmatrix}\text{file\#,}\\ \\ \text{disk-address,}\end{bmatrix} \left[\left(\begin{bmatrix}\text{old-file-name}\\ \text{space}\end{bmatrix}\right)\right] \begin{bmatrix}!\\ \text{P}\end{bmatrix} \text{new-file-name|start-line-no.||,|end-line-no.||}$$

where:

<S>	= A parameter specifying that unnecessary spaces (not including spaces in character strings enclosed in quotes, REM or % statements) will be deleted from a program as it is saved.
<SR>	= A parameter specifying that both spaces and remarks (REM statement lines) are deleted from the program as it is saved.
space	= An expression whose value equals the number of sectors to reserve in addition to the number required to store the program.
old-file-name	= The name of a currently scratched program or data file to be overwritten.
!	= Protect (scramble) the file to be saved.
P	= Set the protection bit on the file to be saved.
new-file-name	= The name of the program to be saved.
start-line-number	= The first line of program text to be saved.
end-line-number	= The last line of program text to be saved.

Purpose:

The SAVE statement causes a BASIC program, or a portion of a program, to be recorded on the designated disk platter. The file name, file type (program file), starting sector address, and ending sector address are entered in the Catalog Index, and the program is automatically stored, starting in a location determined by the system on the basis of the current entries in the Catalog Area.

The '$' parameter specifies that a read-after-write verification test be performed to ensure that all program text is written correctly to the disk. The read-after-write check he time required for the SAVE DC operation, however.

When saving a program on disk, nonessential spaces and/or remarks can be deleted from a program as it is saved by using the '< >' parameter. An <S> implies that any unnecessary spaces (except spaces in character strings enclosed in quotes and % or REM statements) are to be deleted.

NOTE:

The system automatically inserts spaces after line numbers, statement separators (colons), and most BASIC words for readability when displaying a program line. These extra spaces are not in the program in memory.

Using <SR> in the SAVE statement means delete both spaces and remarks (REM statement lines). An <SR> causes all REM statements to be deleted from the program. If certain REM's are to be saved, the REM statements can be changed to image statements %. Image statements are not disturbed by <SR>.

Inclusion of the 'space' parameter instructs the system to reserve a number of sectors in addition to the number actually needed to store the program at the end of the program file. These additional sectors can be used for future expansion of the program. The truncated value of 'space' equals the number of extra sectors to be reserved.

Overwriting an Old File

A new program also can be stored over a scratched program or data file on the disk if the 'old-file-name' parameter is used. The 'old-file-name' parameter specifies the name of the scratched file, and the 'new-file-name' parameter indicates the name of the new program which is to be stored in its place. If the scratched file identified by 'old-file-name' does not occupy adequate space to hold the new program, an error results. When replacing an old program with a new one on disk, it is possible for 'old' and 'new' to be identical. Otherwise, 'new' must be unique (i.e., not already present in the Catalog Index).

If neither the 'old-file-name' nor the 'space' parameter is included in the SAVE statement, the system uses only the exact number of sectors required for the program being stored, and appends the new program file at the current end of the Catalog Area.

If the 'old-file-name' is the same as the 'new-file-name', the 'old-file-name' can be omitted, although the '( )' is still required to indicate that an old file is to be overwritten.

For example,

```
SCRATCH F "RIDER"
SAVE F( ) "RIDER"
```

saves the program currently in memory on the 'F' platter where the old file called "RIDER" was located.

Protecting a Program

The 'P' parameter permits a program to be protected against accidental modification. After a program which has been saved via SAVE P is loaded into memory, the program cannot be modified (except by overlaying). The operator cannot, then, inadvertently modify the program. SAVE P also prevents the program from being listed or resaved.

The '!' parameter performs not only the same function as 'P' but also provides a secure means of preventing program examination. The entire program is scrambled when recorded on disk, and cannot be examined via DATALOAD BA.

The error message display for protected programs is modified to suppress the display of program text. Only the line number, statement separators (:), and error code are displayed.

Example:

```
100 X = 1: Y = 2: Z = 21/0

RUN (EXEC)
```

Output: (unprotected program)

```
100 X = 1: Y = 2: Z = 21/0
                        ↑ERR C62
```

Output: (protected program)

```
100:: ERR C62
```

> **NOTE:**
>
> In order to save or list any program after a protected program has been loaded, it is necessary to clear all of memory either by executing a CLEAR command with no parameters, or by MASTER INITIALIZING the system.

Saving Part of a Program

The 'start-line-no' and 'end-line-no' parameters specify the first and last lines, respectively, of the program in memory which is to be saved. Both of these parameters are optional; if only 'start' is included in the SAVE command, all program lines in memory beginning with that line through the end of the program are saved on disk. If only the 'end-line-no' is specified, all program text from the beginning of the program through the specified line is saved. If neither line number is specified, all program text in memory is saved.

Examples of valid syntax:

```
    SAVE DC F "CONVERT"
    SAVE <S> R "OUTPUT" 300, 500
10  SAVE T $ #2 (100), "OUTPUT2"
10  SAVE F/320, (A$) B$
10  SAVE <SR> T #A, "COORD"
10  SAVE F ("OLD") "NEW"
10  SAVE FP "PROG 1"
10  SAVE R! "MONEY"
10  SAVE T/D11, "PAYLOAD"
```

SCRATCH

General Form:

```
                          ┌ file#,          ┐
  SCRATCH platter         │                 │ file-name [,file-name]• • •
                          └ disk-address,   ┘
```

where:

file-name = The name of the cataloged file (program or data) to be scratched from the catalog.

Purpose:

The SCRATCH statement is used to set the status of the named disk file(s) to a scratched condition. The SCRATCH statement does not remove the files from the catalog; a subsequent listing of the catalog shows the normal information for both scratched and non-scratched files, as well as which files have been scratched. The program text or data in the scratched files is not altered or destroyed by the SCRATCH statement. Once files have been scratched, they cannot be accessed by DATALOAD DC OPEN or LOAD statements. They can, however, be renamed by DATASAVE DC OPEN statements or SAVE commands and statements, and the sectors utilized by scratched files can be reused to save new programs or data files.

The SCRATCH statement is generally used prior to MOVE statements. When a MOVE statement is executed, information concerning all scratched files is deleted from the Catalog Index, and the corresponding program text or data is deleted from the Catalog Area (see MOVE).

NOTE:

Until a MOVE is executed, all scratched file names remain in the Catalog Index, even if the space occupied by the files in the Catalog Area has been renamed and reused. In the latter case, the scratched file name no longer appears in a listing of the Catalog Index, but it continues to occupy space in the Index. A scratched file name is removed from the Index only when a MOVE is executed.

Examples of valid syntax:

```
     SCRATCH F "HEADER"
     SCRATCH R #2, "FLD4/15", "FLD10/7"
     SCRATCH R/320, "COLHDR"
10   SCRATCH F A$, B$, C$
10   SCRATCH F #2, "TEMP 1", A$
10   SCRATCH F #A2, "SORT", "MERGER"
10   SCRATCH T/D12, "HEADER"
10   SCRATCH T #10, "PROB-1", "PROB-2"
10   SCRATCH T "MATTEST"
```

General Form:

```
                        [ file #,        ]
SCRATCH DISK platter    [                ]  [LS = expression – 1,] END = expression – 2
                        [ disk-address,  ]
```

where:

LS	= A parameter specifying the number of sectors to be set aside for the Catalog Index.
expression – 1	= An expression whose value is from 1 to 255. If the 'LS" parameter is not included, the size of the Catalog Index is set automatically at 24 sectors.
END	= A parameter specifying the last (highest) sector address in the Catalog Area.
expression – 2	= An expression whose value must be less than or equal to the last (hightest) sector address on the disk.

Purpose:

The SCRATCH DISK statement is used to reserve space for the Catalog Index and Catalog Area on a disk platter (each disk platter must be initialized separately) prior to saving program files or data files on the disk. This space must be reserved prior to the use of any other catalog statement; otherwise, an error is indicated. Caution must be exercised when using SCRATCH DISK since this statement will destroy an existing Catalog Index on the platter being scratched.

When the SCRATCH DISK statement is executed, the system reserves a number of sectors, starting with sector number 0 on the specified platter, for a disk catalog. The 'LS' parameter defines the size of the Catalog Index, and the value of 'expression-1' specifies the number of sectors to be reserved. A maximum of 255 sectors (sectors 0-254) can be reserved for the Index. If the 'LS' parameter is not included in the SCRATCH DISK statement, 24 sectors (sectors 0-23) are reserved automatically for the Index. The entry for each cataloged file in the Catalog Index consists of the file's name and associated sector address parameters; each sector of the Index can hold 16 file entries, with the exception of sector 0, which holds 15 entries (a small portion of sector number 0 contains systems information used to maintain the catalog). When the catalog is initially established, the remainder of sector number 0 and all other sectors reserved for the Catalog Index are filled with zeroes.

The END parameter defines the limit of the Catalog Area on disk. The value of 'expression-2' specifies the address of the last sector to be used for storing cataloged files. The END parameter is particularly useful when temporary work files are to be established since temporary files must be established outside the Catalog Area. An error will result if the user attempts to establish a temporary file within the Catalog Area.

The end of the Catalog Area can be altered with the MOVE END statement (see MOVE END).

> **NOTE:**
>
> Although, in general, the Catalog Area can be expanded or retracted when necessary with the MOVE END statement, the size of the Catalog Index cannot be altered once specified without reorganizing the entire catalog. Take special care, therefore, to provide ample space for future expansion when specifying the size of the Catalog Index in the 'LS' parameter. Number of entries in Index = 16 (no. of index sectors)-1.

Examples of valid syntax:

```
    SCRATCH DISK R END = 9791
    SCRATCH DISK F LS = 4, END = 1000
100 SCRATCH DISK F/320, END = X*2
100 SCRATCH DISK T #X, LS = L, END = E
100 SCRATCH DISK T/B10, END = 10000
100 SCRATCH DISK T END = 12550
```

General Form:

```
                 ┌ file #,      ┐
VERIFY platter   │              │ [ (start-sector, end-sector) ] [ numeric-variable ]
                 └ disk-address,┘
```

where:

start-sector	= An expression whose value equals the address of the first sector to be verified.
end-sector	= An expression whose value equals the address of the last sector to be verified.
numeric-variable	= A numeric variable designated to receive a value equal to the address + 1 of the first spector that did not verify during the verify operation (equal to zero if no errors).

Purpose:

The VERIFY statement reads all sectors within the specified range from the designated disk platter, and performs cyclic and longitudinal redundancy checks to ensure that information has been written correctly to those sectors. The value of 'start-sector' specifies the address of the first sector to be verified, and the value of 'end-sector' specifies the address of the last sector to be verified. If the 'start-sector' and 'end-sector' parameters are omitted, the entire Catalog Index and Catalog Area (up to the current end) are verified.

When an error is detected during the verify operation, one of two things happens. If a numeric-variable is specified, the variable is set equal to the sector-address +1 of the sector in error and the verify operation terminates. (If there were no errors, the numeric-variable is set equal to zero.) If the numeric-variable is omitted, an error message showing the sector address of the bad sector is displayed on the Console Output device and the verify operation continues. The HALT/STEP key can be used to terminate the printout of erroneous sectors, but the verify operation cannot be continued.

Examples of valid syntax:

```
10  VERIFY F #2,
10  VERIFY T #A(1), (100,200)
10  VERIFY R (0,1023)
10  VERIFY F/320, (0,2000)
10  VERIFY F #15,(50,300)R
10  VERIFY T/D14, (0,10550)
10  VERIFY T/D11,
```

```
ERROR OUTPUT:
   ERROR IN SECTOR 1097
   ERROR IN SECTOR 8012
```

$FORMAT DISK

General Form:

$$\text{\$FOR MAT DISK Platter} \left\{ \begin{array}{l} \text{file \#} \\ \\ \text{disk-address} \end{array} \right\}$$

Purpose:

The $FORMAT DISK statement is used to format a disk platter. Before a platter can be used for the storage and retrieval of data, the platter must be formatted. Formatting involves recording a unique address for each sector on the disk platter, along with other control information. The control information helps the system maintain the disk and keep a check on the validity of the data written to and from it.

CAUTION:

Formatting a disk platter overwrites all data previously stored on the platter. Zeros are written within each sector.

The $FORMAT DISK statement is available in 2200VP BASIC-2 Release 1.9, 2200MVP BASIC-2 Release 1.7, and subsequent editions. It can only be used with Wang disk units that support formatting under software control (e.g., Models 2260C, 2260BC, and 2280).

Examples of valid syntax:

```
10  $FORMAT DISK T/310
20  $FORMAT DISK T/D11
30  $FORMAT DISK R#2
```

CHAPTER 6
ABSOLUTE SECTOR ADDRESSING

6.1 INTRODUCTION

Absolute Sector Addressing Mode is comprised of eight BASIC-2 statements and commands which enable the programmer to read or write information in specific sectors on the disk. No catalog or Catalog Index can be established or maintained in Absolute Sector Addressing Mode (except by user-supplied software), nor is it possible to name programs or data files. Files are identified only by reference to their starting sector addresses. Similarly, individual records must be saved into or loaded from a file by specifying a starting sector address. All file addressing information must be maintained by the programmer; such information is not maintained automatically by the system. Because the disk statements in Absolute Sector Addressing Mode provide direct access to individual sectors, they are referred to as "direct addressing" statements.

It is sometimes useful to use direct addressing statements in conjunction with automatic file cataloging statements. The DC statements can provide a framework for file access and control while the direct addressing statements provide the programmer with a means of writing customized disk operating systems and special access procedures within a file, such as binary searches, sorting routines, etc. which cannot be done efficiently - and, in some cases, which cannot be done at all - with catalog procedures alone. Two classes of statements are available in Absolute Sector Addressing Mode: the DA statements (where "DA" is a mnemonic for "direct address") and the BA statements (where "BA" is a mnemonic for "block address"). Both permit direct access to specific sectors on the disk.

The DA statements can be used to write or read programs or data records beginning at a specified sector on the disk. Multi- sector programs and data records are automatically read or written, just as they are with DC statements. All records saved with a DA statement or command are automatically formatted to contain the standard System 2200 control information (see Chapter 4, Section 4.6), and records loaded with a DA statement or command must contain this format information. Records created by DA statements or commands are, therefore, identical in format to records created by DC (catalog) statements or commands, and records saved in one mode may be retrieved in the other.

The BA statements comprise a special class of statements which read and write exactly one sector (256 bytes) of unformatted data. Records created with a DATASAVE DC or DATASAVE DA statement are automatically formatted by the system to contain certain control information. (Refer to Chapter 4, Section 4.6, for a discussion of the control information automatically included in each sector of a logical record.) When a data record is read from the disk with a DATALOAD DC or DATALOAD DA statement, the system expects to find the control information; a record which does not contain the expected control information cannot be read with a DC or DA statement. When a record is created with a DATASAVE BA statement, however, no control information is written by the system. In this special case, the programmer is free to write his own control information in each record, and to format his records in a way best suited for his application. Records with a non-standard format can be read with a DATALOAD BA statement; they cannot be read with DC or DA statements. DATALOAD BA can also be used to read sectors (program or data) written originally with a DC or DA statement.

No Catalog or Catalog Index is established or maintained in Absolute Sector Addressing Mode and the Device Table is used only to obtain the disk-address.

The DA and BA statements do not modify any file start, end, or current sector address information in the Device Table.

In addition to reading and writing information on the disk, Absolute Sector Addressing Mode also provides the capability to perform platter-to-platter copy operations and verify the transferred data. The Absolute Sector Addressing statements and commands are:

SAVE DA
LOAD DA (command)
LOAD DA (statement)
DATASAVE DA
DATALOAD DA
DATALOAD BA
COPY
VERIFY

6.2 SPECIFYING SECTOR ADDRESSES

When a data record or program is saved or loaded with a direct addressing statement or command, the starting sector address must be specified by the programmer. The address may be supplied in the form of an expression, or as the value of an alphanumeric variable. If the address is supplied as the value of an alpha variable, the binary value of the first two bytes of that variable is interpreted as the sector address. The value of the expression or alpha variable must, of course, be less than or equal to the last (highest) sector address on the disk platter. After the statement is processed, the system automatically returns the address of the next available sector. A second alpha or numeric variable can be included in the statement to receive this address.

```
SAVE DA F  (100,             A)
           ↑                 ↑
           Specifies the     After execution of
           address of the    the SAVE DA command,
           first sector on   this variable contains
           the 'F' platter   the address of the next
           to be used to     available sector on the
           store the         'F' platter.
           saved program.
```

In order to economize on the use of memory and disk space, and to facilitate address calculations in binary, the beginning sector address and the next available sector address may be expressed as two-byte binary values, i.e., as the first two bytes of alphanumeric variables (see Example 6-8). A sector address expressed as a two-character binary number occupies only two bytes of memory or three bytes of disk storage, while the same address expressed as a decimal value requires eight bytes of memory and nine bytes of disk storage. The savings in storage space gained by expressing the sector address in binary can become appreciable when, for example, key files are established to facilitate random access operations. Typically, a key file contains a list of keys along with the sector addresses of records identified by those keys. In a key file containing, say, 9,000 keys and sector addresses, some 7,000 bytes of disk storage (about 27 sectors) are saved by expressing the sector addresses in binary rather than decimal. If the starting sector address is to be expressed as a binary number, it must be specified as the value of an alphanumeric variable (the first two bytes are used). If the next available sector address is to be returned as a binary number, the receiving variable must be specified as an alphanumeric variable of at least two characters in length.

6.3 STORING AND RETRIEVING PROGRAMS ON DISK IN ABSOLUTE SECTOR ADDRESSING MODE

In Absolute Sector Addressing Mode, the programmer himself must keep track of each program's location on the disk. The starting sector address of the program must be directly specified by the programmer when writing or reading a program on disk; it becomes the responsibility of the programmer, therefore, to ensure that information already recorded on disk is not overwritten by each new program, and that the location of each program is saved for future reference. Because there are few cases in which the advantage to be gained in direct addressing program operations offsets the added complexities involved, program storage and retrieval are not commonly done in Absolute Sector Addressing Mode.

Apart from the important fact that a direct addressing statement must specify an absolute sector address rather than a file name, the SAVE DA and LOAD DA instructions are not remarkably different from their cataloging counterparts, SAVE [DC] and LOAD [DC]. Specifically, the format of a program file written on disk with SAVE DA is almost identical to that of a cataloged file written with SAVE DC. In both cases, the program file begins with a one-sector header record, and ends with a trailer record. (In cataloged program files, the header record contains the file name; in program files created with SAVE DA, the header record contains a field of blanks in place of a file name.) An additional sector of control information, the end-of- file control record, is written at the end of every cataloged program file by SAVE [DC]. This control record is not written in program files recorded with SAVE DA.

The close similarity between the formats of cataloged program files and those created with direct addressing statements makes it possible for programs recorded in catalog mode (with SAVE DC) to be read in direct addressing mode (with LOAD DA). LOAD DA, like LOAD [DC], begins reading a program at the header record (the starting sector address of the program must, therefore, be known), and terminates reading when it encounters the trailer record. In this way, the entire program file is automatically read and loaded into memory. In normal operations, there is no advantage to be gained by loading cataloged programs with LOAD DA; it is generally safer and easier to use LOAD DC. The only situation in which it could be advantageous to employ LOAD DA for cataloged program files is recovery from an accident which destroys entries for one or more program files in the Catalog Index, without harming the programs themselves. In such a situation, the programs can be accessed only with direct addressing.

The LOAD [DC] statement cannot be used to read non- cataloged files recorded with SAVE DA. SAVE DA does not record the file name and sector address parameters in the Catalog Index when a file is saved, and LOAD DC cannot access a program without this information.

Saving Programs on Disk with SAVE DA

Programs are stored on disk in Absolute Sector Addressing Mode with a SAVE DA command. The following items of information must be included in the command:

1. The disk platter on which the program is to be stored.
2. The address of the first sector on the disk in which the program is to be stored (specified as an expression or alphanumeric variable).
3. Optionally, numeric or alphanumeric return variable designated to receive the address of the first free sector following execution of the SAVE DA command.

4. Optionally, one or two line numbers identifying the program lines which are to be saved on disk. If one line number is specified, all program lines beginning at that line are saved on disk. If two line numbers are specified, all program lines between and including those two lines are saved. If no line number is specified, all resident program text is saved.

Example 6-1: Saving Program on Disk with SAVE DA (No Line Numbers Specified)

```
SAVE DA F (1250,L)
```

This command (SAVE DA is also programmable) causes all program lines currently in memory to be saved on the 'F' disk platter, beginning at sector 1250. As many sectors are used as are needed to store the resident program text. Following execution of the command, the address of the next available sector is returned to numeric variable L as a decimal value. For example, if the program required 10 sectors on disk (sectors 1250-1259), then L = 1260 following execution of the command.

Example 6-2: Saving a Program on the Model 2280 Disk with SAVE DA (No Line Numbers Specified)

```
SAVE DAT/D13, (2970,M)
```

This command causes all program lines currently in memory to be saved on the D13 fixed platter, beginning at sector 2970. As many sectors are used as are needed to store the resident program text. Following execution of the command, the address of the next available sector is returned to numeric variable M as a decimal value. For example, if the program required 125 sectors on disk (sectors 2970-3094), then M = 3095 following execution of the command.

Example 6-3: Saving a Program on Disk with SAVE DA (Two Line Numbers Specified)

```
SAVE DA R (1300,N) 100, 750
```

SAVE DA causes lines 100 through 750 to be recorded on the 'R' platter starting at sector 1300, and uses as many sectors as it needs to store the program. When the program is recorded, the address of the next available sector is returned to variable N.

Example 6-4: Saving a Program on the Model 2280 Disk with SAVE DA (Two Line Numbers Specified)

```
SAVE DA T/D12, (4690,B) 50, 910
```

SAVE DA causes lines 50 through 910 to be recorded on the D12 fixed platter starting at sector 4690, and uses as many sectors as it needs to store the program. When the program is recorded, the address of the next available sector is returned to variable B.

A useful feature of the SAVE DA statement is the ability to delete unnecessary spaces and remark statements in a program before it is stored on the disk. If the parameter <S> is specified in the SAVE DA statement all unnecessary spaces (excluding spaces in character strings enclosed in quotes and in % or REM statements) will be deleted from the program as it is saved. By specifying the parameter <SR> in the SAVE DA statement both unnecessary spaces and remarks (REM statements) are removed from the program when it is saved.

Example 6-5: Saving a Program on Disk with SAVE DA Using the <S> Parameter

```
10 REM THIS IS PART 5 OF PROGRAM PAYROLL
20 PRINT "CALCULATE X,   Y"
30 X = 3.14: Y = 5*X+7:   GOTO 100
  .
  .
  .
100 REM LOAD PART 6
SAVE DA <S> R (1500)
```

SAVE DA causes the program (lines 10 through 100) to be recorded on disk starting at sector 1500, and uses as many sectors as is required to store the program. The <S> deletes the unnecessary spaces in line 30 but does not effect the spaces in line 10 or the spaces enclosed in quotes in line 20. If the parameter <SR> is used in SAVE DA instead of <S>, then program lines 10 and 100 are deleted in addition to the unnecessary spaces in line 30.

Retrieving Programs from Disk with LOAD DA

The LOAD DA instruction, like its catalog counterpart LOAD DC, is a hybrid having two distinct forms, the LOAD DA command and the LOAD DA statement. As with LOAD [DC], the two forms of LOAD DA have significantly different functions, and must be discussed separately. In both forms of LOAD DA, however, the starting sector address of the program to be loaded must be specified. It is important to note in this context that LOAD DA always expects to read a complete program, beginning with a header record, including one or more program records, and ending with a trailer record. For this reason, it is not possible to begin program loading in the middle of a program, or at any point beyond the program header record. For example, if the starting sector address of a program is sector #100, and the starting address specified in a LOAD DA instruction is 101 or beyond, the program is not loaded.

The LOAD DA Command

The LOAD DA command causes a program to be read from disk, beginning at a specified sector address, and appended to existing program text in memory. Program lines in memory having the same numbers as lines in the newly loaded program are cleared and replaced by the new lines. Resident program lines with different line numbers are not cleared, however, and remain in memory following the LOAD DA operation. For this reason, resident program text should generally be cleared with a CLEAR or CLEAR P command prior to loading in the new program.

Example 6-6: Loading a Program from Disk with LOAD DA Command

```
CLEAR
LOAD DA F (100, D)
```

The LOAD DA command causes the system to load a BASIC program from the 'F' platter beginning at sector 100 (if sector 100 does not contain a program header record, an error results). When the program has been loaded, the address of the next sequential sector following the trailer record is returned to variable D. (For example, if the program trailer record resides at sector #112, D = 113 following execution.)

Example 6-7: Loading a Program from the Model 2280 Disk with LOAD DA Command

```
CLEAR
LOAD DA T/D10, (250, P)
```

The LOAD DA command causes the system to load a BASIC program from the D10 removable platter beginning at sector 250 (if sector 250 does not contain a program header record, an error results). When the program has been loaded, the address of the next sequential sector following the trailer record is returned to variable P. (For example, if the program trailer record resides at sector #275, P = 276 following execution.)

The LOAD DA Statement

The operation of the LOAD DA statement is analogous to that of the LOAD DC statement. LOAD DA permits programs to be loaded from a specified sector location on disk under program control. Prior to loading the program from disk, LOAD DA automatically clears out all or a specified portion of the resident program text, as well as all noncommon variables. (Common variables are not cleared.) Once loaded in memory, the new program is executed automatically.

The LOAD DA statement contains the following parameters:

1. A platter parameter ('F', 'R' or 'T').
2. The starting sector address of the program to be loaded, specified as an expression or alpha variable. This address must be the address of the program header record.
3. Optionally, numeric or alphanumeric return variable designated to receive the address of the next sequential sector following the program trailer record. (Note: This variable must be a common variable.)
4. Optionally, one or two program line numbers defining the portion of resident program text which is to be cleared prior to loading the new program. Inclusion of one line number causes all program lines beginning at that line to be cleared. Inclusion of a pair of line numbers causes all program lines between and including the two specified lines to be cleared. Omission of both line numbers causes the totality of resident program text to be cleared.
5. Optionally, a third program line number preceded by 'BEG'. This line number value signifies the line number of the program where execution is to begin after the program is loaded into memory.

Example 6-8: Loading Programs from Disk with a LOAD DA Statement (No Line Number Specified)

```
10 COM D
50 LOAD DA F (24,D)
```

Statement 50 causes a program to be loaded from the 'F' platter beginning at sector 24. Prior to loading in the new program, all program text in memory is cleared, along with all non-common variables. After the new program has been loaded, program execution begins automatically at the lowest program line. The address of the next sequential sector following the program trailer record is returned as a decimal value to numeric variable D. For example, if the program trailer record is located in sector #33, then D = 34 following execution of statement 50.

Note that the return variable ('D' in the above example) must be a common variable. Otherwise, it is cleared along with all other noncommon variables before the program is loaded, and an Error P54 (Common Variable Required) is signalled.

The LOAD DA statement, like LOAD [DC], can be used to load program overlays from disk and append them to an existing program in memory. In this case, one or both of the optional line number parameters are specified to define the portion of resident program text which must be cleared prior to loading the program overlay. Note that when one or both line numbers are included, execution of the overlay begins automatically at the first line number specified in the LOAD DA statement. If the new program does not contain a line having the first line number specified, an ERROR P36 (Missing Line Number) is signalled.

If the program overlays are stored in sequential areas of the disk, it is possible to use the same variable to contain the starting sector address and receive the address of the next available sector following statement execution. In this way, the starting sector address is automatically updated to the address of the next available sector every time the LOAD DA statement is executed. Note that this technique must be modified if cataloged programs are read, since a cataloged program has an additional system end-of-file sector following the trailer record which is not read as part of the program by LOAD DA, and the address of this sector will be returned by the LOAD DA statement. For normal processing, it is recommended that cataloged programs be read only with the catalog instruction LOAD DC.

Example 6-9: Loading Program Overlays from the Disk with the LOAD DA Statement (Two Line Numbers Specified)

```
80  COM D
90  D = 24
.
.
.
500 LOAD DA F (D,D) 100,490
```

Statement 500 causes a program to be loaded into memory from the 'F' platter, starting at the sector whose address is stored in D. Prior to loading the program overlay, program lines 100 through 490 are cleared from memory, along with all non-common variables. After the program has been loaded, program execution begins automatically at line 100. Following statement execution, the address of the next available sector is returned to D (however, D must have been specified as a common variable). When Statement 500 is executed a second time, the new value of D is the starting sector address of the second program overlay (assuming that the overlays are stored sequentially on the disk, and that they are not cataloged files). The second overlay is automatically loaded over the first overlay, and run from line 100. The process can be continued in this way for as long as necessary.

Example 6-10: Loading a Program from the Disk with LOAD DA Statement (BEG Line Number Specified)

```
100 LOAD DA R (70) BEG 120
```

Statement 100 causes a program to be loaded into memory from the 'R' platter, starting at sector 70. Prior to loading the new program, all program text in memory is cleared, along with all non-common variables. After the new program has been loaded, program execution begins automatically at line 120.

Example 6-11: Loading a Program from the Model 2280 Disk with LOAD DA Statement (BEG Line Number Specified)

```
100 LOAD DA T/D10, (165) BEG 90
```

Statement 100 causes a program to be loaded into memory from the D10 platter, starting at sector 165. Prior to loading the new program, all program text in memory is cleared, along with all non-common variables. After the new program has been loaded, program execution begins automatically at line 90.

6.4 STORING AND RETRIEVING DATA ON DISK IN ABSOLUTE SECTOR ADDRESSING MODE

In Absolute Sector Addressing Mode, named data files are not maintained by the system, nor are the file parameters stored in the Catalog Index or Device Table.

Storing Data on the Disk with DATASAVE DA

Data is stored on the disk in Absolute Sector Addressing Mode with the DATASAVE DA statement. At least four items of information must be provided in the statement:

1. The disk platter on which the data is to be saved.
2. The address of the first sector on that platter in which the data is to be stored (specified as an expression or alphanumeric variable).
3. Optionally, numeric or alphanumeric variable which is to receive the address of the next available sector following statement execution.
4. The data which is to be saved in a record on the disk.

Each DATASAVE DA statement (like DATASAVE DC) saves one logical record, consisting of enough sectors on disk to store all data specified in the argument list. Records saved on the disk with DATASAVE DA are identical in format to those created by DATASAVE DC, and contain the standard System 2200 format information. Records initially saved with DATASAVE DA can therefore be loaded with DATALOAD DC. Note, however, that when DATASAVE DA is used to write a record in a cataloged data file, it does not update the file parameters in the Catalog Index. In normal processing operations, the use of direct addressing statements to alter cataloged files is not recommended.

Example 6-12: Storing Data on Disk with a DATASAVE DA Statement

```
100 B$ = HEX(01E0)
150 DATASAVE DA R (B$,X$) A, B(), C()
```

Statement 150 causes the value of numeric variable A and arrays B() and C() to be stored on the 'R' platter, starting at sector #480 (480 is the decimal equivalent of HEX(01E0), which is the value of B$). One logical record is written containing enough sectors to store all data specified in the argument list. Following the execution of statement 150, X$ is set equal to the binary address of the next available sector. For example, if A, B(), and C() require nine sectors on the disk, the value of X$ following statement execution is HEX(01E9) (decimal equivalence, 489).

Example 6-13: Storing Data on the Model 2280 Disk with a DATASAVE DA Statement

```
100 D$ = HEX(00FF)
200 DATASAVE DA T/D11, (D$,F$) A, B( ), C( )
```

Statement 200 causes the value of numeric variable A and arrays B( ) and C( ) to be stored on the D11 platter, starting at sector #255 (255 is the decimal equivalent of HEX(00FF), which is the value of D$). One logical record is written containing enough sectors to store all data specified in the argument list. Assuming that A, B( ), and C( ) require 15 sectors on the disk, the value of F$ following statement execution is HEX (010E) (decimal equivalence, 270).

If a number of records are to be saved or loaded in sequential sectors on the disk, it is possible to use the same variable to contain the starting sector address and receive the address of the next available sector following statement execution. In this way, the starting sector address is automatically updated to the address of the next available sector following each save or load operation.

Example 6-14: Saving a Number of Data Records in Sequential Areas of the Disk

```
200 DIM B(25)
210 A1 = 50
220 FOR I = 1 TO 25
230 INPUT "VALUES FOR THIS RECORD", B(I)
240 NEXT I
250 DATASAVE DA F(A1,A1) B( )
:
:
290 GOTO 220
```

The starting sector address (A1) is initially set to 50. At line 230, the values to be stored in the first record are entered. The first time through the loop, line 250 saves array B( ) on the 'F' platter beginning at sector 50. When the record has been written, the address of the next available sector is returned in A1. Assuming that B( ) required ten sectors, A1 is set equal to 60 following execution of statement 250. The second time through the loop, array B( ) is saved on the 'F' platter beginning at sector 60, since this is the new value of A1. The process may be continued in this way in order to store records in sequential areas of the disk.

After all data records have been saved in a file, the file should be ended with an end-of-file trailer record, which can be used subsequently to test for the end-of-file if the records are read sequentially for processing. In Absolute Sector Addressing Mode, the trailer record is the programmer's only way of protecting himself from reading beyond the legitimate data in a file (unless he designs his own trailer record), since the data file has no absolute limit (as it does in catalog mode). If no trailer record is written, the program may read beyond the limit of legitimate data in the file and retrieve meaningless data from the subsequent, unused sectors. An end-of-file record is written in Absolute Sector Addressing Mode exactly as it is written in Catalog Mode, by specifying the "END" parameter instead of an argument list in a DATASAVE DA statement:

Example 6-15: Writing an End-Of-File Record in a Data File with a DATASAVE DA END Statement

```
180 DIM B(25): A1 = 50
190 FOR I=1 TO 25
200 INPUT "VALUES FOR THIS RECORD", B(I)
210 NEXT I
220 DATASAVE DA R (A1,A1) B()
230 INPUT "IS THIS THE LAST RECORD? (Y OR N)", F$
240 IF F$ = "Y" THEN 350
250 GOTO 190
:
:
350 DATASAVE DA R (A1,A1) END
```

This routine illustrates a simple input loop in which the operator is asked after entering each record if it is the last record. If a response of "N" (or any response other than "Y") is entered, the routine loops back to input another record. When a response of "Y" is entered, however, the routine branches to line 350 and writes an end-of-file record in the file.

When a new record is written into a file which has been ended with a trailer record, the trailer record should be overwritten, and a new trailer record created following all subsequent data saving operations. For example, if the trailer record occupies sector 497 in a file, the next data record should be saved beginning at sector 497, and a new trailer record written following the save operation.

Retrieving Data from Disk with DATALOAD DA

Data is retrieved from a data file on the disk in Absolute Sector Addressing Mode with a DATALOAD DA statement. Four items of information must be specified:

1. The disk platter on which the data is stored.
2. The address of the first sector on that platter from which data is to be read (specified as an expression or alphanumeric variable).
3. Optionally, a numeric or alphanumeric return variable designated to receive the address of the next sequential logical record following statement execution.
4. An argument list consisting of one or more alpha or numeric receiving variables, arrays, or array elements designated to receive the data read from the disk.

Example 6-16: Retrieving Data from a Data File on Disk with a DATALOAD DA Statement

```
300 DATALOAD DA R (481,B2) A,B,C
```

Statement 300 causes the system to load data from the 'R' platter beginning at sector 481, and store the data in numeric variables A, B, and C in memory. Enough data is read from the disk to fill all variables specified in the argument list (unless the trailer record is encountered, at which point reading stops). However, it is recommended that exactly one logical record be read with each DATALOAD DA statement. In order to read one logical record, the argument list of the DATALOAD DA statement must correspond to the argument list of the DATASAVE DA statement which originally saved the record. If only the first few fields in a logical record are loaded, the remaining fields in the record are read but ignored. If the argument list contains more receiving arguments than there are fields in a logical record, values are read from the next sequential logical record until the argument list is filled. The remainder of the second record is then read and ignored. Following statement execution, the return variable B2 is set to the address of the next sequential logical record. Thus, if the record occupies three sectors (481, 482, 483), B2 = 484 following statement execution.

If an end-of-file record has been written in the data file, it is possible to test for the end-of-file condition with an IF END THEN statement. The IF END THEN statement is useful when processing records sequentially from a file since it terminates reading and initiates a branch to a specified line number when the end-of-file record is read. The end-of-file record is not transferred into the DATALOAD DA argument list, and the value of the return variable in the DATALOAD DA statement is set to the address of the end-of-file record rather than to the next sequential sector. The system is therefore positioned to save a new record over the end-of-file record if additional data is to be stored in the file.

Example 6-17: Testing for the End-of-File Condition in a Non-Cataloged Data File

```
400 DATALOAD DA R (B2,B2) A()
410 IF END THEN 500
:
:
490 GOTO 400
500 STOP
```

Statement 400 loads one logical record from the 'R' platter, beginning at the sector whose address is stored in B2, and stores the data in array A( ). Statement 410 checks for an end-of-file trailer record (previously written with a DATASAVE DA END statement). If the trailer record is detected, the program skips to statement 500 and stops. If no trailer record is detected, program execution continues normally, with data in A( ) being processed until, at statement 490, the system is instructed to loop back and load in another record. Note that when the trailer record is read, the receiving variable (B2) is set to the address of the trailer record, not the address of the next consecutive sector.

6.5 THE 'BA' STATEMENTS

Two special statements, DATASAVE BA and DATALOAD BA, enable the programmer to save and load records that do not contain the standard System 2200 control information (such records cannot be saved or loaded with DC or DA statements). Since records saved or loaded with a BA statement are not formatted automatically with System 2200 control information, the programmer is free to write his own control information, and format his records in a manner appropriate to his application. Records which are saved with a DATASAVE BA statement must be loaded with a DATALOAD BA statement. The DATALOAD DC and DATALOAD DA statements cannot be used to read a record which was saved initially with DATASAVE BA. However, DATALOAD BA can be used to read sectors which were written initially with DC or DA statements or commands.

The DATASAVE BA statement writes exactly one sector (256 bytes) of unformatted data into a specified sector on the disk. A single variable, alpha array, or literal-string must be used in the DATASAVE BA argument list. Numeric variables and arrays, are illegal. Multiple arguments are not permitted. It is not possible to write a multi-sector record with a single DATASAVE BA statement. If the alpha array in the DATASAVE BA argument list contains more than 256 bytes of data, the additional data is ignored. If the array contains fewer than 256 bytes, the remainder of the sector being addressed is filled with unpredictable data. It is therefore always advisable to specify an array which contains at least 256 bytes of data in the DATASAVE BA argument list. Four items of information must be specified in the DATASAVE BA statement:

1. The disk platter on which the data is to be stored.

2. The address of the sector in which the data is to be written (multi-sector records are not written automatically).

3. Optionally, a numeric or alphanumeric return variable designated to receive the address of the next consecutive sector following statement execution.

4. One alphanumeric variable, array or literal containing the data to be saved on the disk. (It is recommended that the array contain 256 bytes of data.)

Example 6-18: Writing an Unformatted Sector with DATASAVE BA

```
200 DATASAVE BA F (L$,L$) A$( )
```

Statement 200 causes 256 bytes of unformatted data to be transferred from array A$( ) into the sector on the 'F' platter whose address is stored in alpha variable L$. If A$( ) contains more than 256 bytes of data, the additional data is ignored. If A$( ) contains fewer than 256 bytes of data, the remainder of the sector is filled with garbage. Following statement execution, the address of the next consecutive sector is returned to L$ (i.e., if L$ = HEX(01E0) prior to execution of statement 200, then L$ = HEX(01E1) following statement execution).

Example 6-19: Writing an Unformatted Sector on the Model 2280 Disk with DATASAVE BA

```
500 DATASAVE BA T/D14, (F$,F$) B$( )
```

Statement 500 causes 256 bytes of unformatted data to be transferred from array B$( ) into the sector on the D14 platter whose address is stored in alpha variable F$.

The DATALOAD BA statement loads exactly one sector (256 bytes) of data from a specified sector on the disk into a specified alphanumeric array in memory (numeric arrays, as well as alpha and numeric variables and array elements, are illegal). The receiving array must be dimensioned to contain at least 256 bytes. If the array contains fewer than 256 bytes, an error is signalled and the data is not transferred; if the array contains more than 256 bytes, the additional bytes are undisturbed. It is not possible to read multi-sector records with the DATALOAD BA statement. The DATALOAD BA statement must include the same four elements specified in the DATASAVE BA statement (i.e., disk platter to be accessed, address of sector to be loaded, variable specified to receive address of next consecutive sector (optional), and alpha array specified to receive data read from disk).

Example 6-20: Reading a Sector from Disk with DATALOAD BA

```
240 DIM A$(256)1
250 DATALOAD BA F (20,L) A$( )
```

Statement 250 causes all information stored in sector 20 on the 'F' platter (256 bytes) to be loaded into alpha array A$( ) in memory. A$( ) is dimensioned at line 240 to contain 256 bytes of data. If A$( ) held fewer than 256 bytes, an error would be signalled. Following execution of the statement, the address of the next consecutive sector is returned in numeric variable L (i.e., following statement execution, L=21). If A$( ) were dimensioned larger than 256 bytes, the additional bytes of A$( ) would remain unaltered.

6.6 PLATTER-TO-PLATTER COPY WITH "COPY"

Absolute Sector Addressing Mode provides the capability to copy all or part of the contents of one disk platter onto another with the COPY statement. The entire contents of a disk platter, or any specified portion of its contents, can be copied from one disk platter to another, and from one disk unit to another disk unit. Unlike MOVE (see the discussion of MOVE in Chapter 2), COPY transfers all information located on the specified portion of the disk platter which is to be copied (including scratched and temporary files) to the corresponding sectors on the second platter. If the beginning and ending sector addresses of the portion of the disk platter which is to be copied are not specified, the disk catalog index and cataloged files are copied. If the disk is not catalogued or if the entire disk platter is to be copied, the starting sector address specified must be 0 and the ending sector address specified must be the highest sector address on the platter. It is recommended that MOVE be used instead of COPY when transferring the catalog from platter to platter (since, in that case, scratched files are automatically deleted).

Example 6-21: Copying Disk Platters in the Same Disk Unit with the COPY Statement

```
10 COPY R (0, 2399) TO F
```

Statement 10 causes the contents of sectors zero through 2399 to be transferred from the 'R' disk platter to the corresponding sectors (0 - 2399) on the 'F' disk platter.

Example 6-22: Copying Disk Platters in the Model 2280 Disk Unit with the COPY Statement

```
100 COPY T/D12, (0, 5800) TO T/D10,
```

Statement 100 causes the contents of sectors zero through 5800 to be transferred from the D12 platter to the corresponding sectors (0 - 5800) on the D10 platter.

If it is convenient, the starting and ending sector addresses may be expressed as the values of numeric variables or expressions.

Example 6-23: Copying from One Disk Unit to Another with COPY (Model 2260 and 2270 Series)

```
700 A=10

710 COPY R/310, (A,A*100) TO F/320,(100)
```

Statement 710 causes sectors 10 (the value of A) through 1000 (the value of A*100) to be transferred from the 'R' platter in the disk unit with address 310 to the 'F' platter in the disk unit on the same system with address 320. The information is copied on the 'F' platter starting with sector 100. All scratched and temporary files are included in the transfer. If the 'starting-sector' (A) and 'ending-sector' (A*100) expressions had been omitted, the catalog and cataloged files would have been copied.

Following a COPY operation, the transferred information should be checked to ensure that it has been written correctly. The VERIFY statement is used to perform such a validation check. If the entire contents of the disk platter are copied, the entire platter can be checked by executing a VERIFY statement which specifies sector 0 as the starting address, and the address of the last sector on the platter as the ending address. If only a specific portion of a platter is transferred, the VERIFY statement can be used to verify only that portion of the second platter.

Example 6-24: Verifying Data Transfer Following a COPY Operation

```
10 COPY F (0,1000) TO R
20 VERIFY R (0,1000)
```

Statement 10 copies sectors zero through 1000 from the 'F' platter to the same sectors on the 'R' platter. Statement 20 verifies the newly-copied sectors 0 - 1000 on the 'R' platter.

Example 6-25: Verifying Data Transfer on the Model 2280 Following a COPY Operation

```
100 COPY T/D14, (4000,8000) TO T/D10,
200 VERIFY T/D10, (4000,8000)
```

Statement 100 copies sectors 4000 through 8000 from the D14 platter to the same sectors on the D10 platter. Statement 200 verifies the newly-copied sectors 4000 - 8000 on the D10 platter.

If the check performed by VERIFY is positive, the system returns the CRT cursor and colon to the screen, indicating that the information has been written accurately. If one or more errors are discovered, the system returns an error message indicating which sector(s) did not verify properly, e.g.,

```
ERROR IN SECTOR 946
```

If you encounter an error following a COPY operation, repeat the COPY. Repeated failure could indicate a faulty disk platter. If the error persists with another platter, call your Wang Service Representative.

VERIFY can be used to verify any portion of a disk platter, or an entire platter, for any reason. It need not be used only in conjunction with COPY. It may be useful, for example, to verify data on a previously recorded platter before the platter is reused. Many programmers verify each platter at the beginning of daily operation. The CRC and LRC checks performed by VERIFY provide an early detection of improperly written information on the disk.

WARNING:

It is important that backup copies of important disk-based data files be maintained and kept up to date. Like other storage media, disk platters can be worn out with repeated use, and they are, of course, subject to accidental damage or destruction. To avoid the necessity of recreating your data base following such a disaster, you should always maintain one or more backup platters containing all important files. Non-cataloged files can be copied to a backup platter with the COPY statement. For cataloged files, the MOVE statement should be used.

6.7 USING ABSOLUTE SECTOR ADDRESSING STATEMENTS IN CONJUNCTION WITH CATALOG PROCEDURES (BINARY SEARCH)

In the concluding paragraph of Chapter 4, it was pointed out that Absolute Sector Addressing statements can be used in conjunction with catalog procedures to develop more versatile and efficient file access techniques. One of the data retrieval techniques most commonly used on data files is the binary search technique. The System 2200 provides a special BASIC verb, LIMITS, which can be used in conjunction with direct addressing statements to perform a binary search on cataloged files. (LIMITS is discussed in Chapter 4, Section 4.7.)

A binary search is a technique for locating a particular record in an ordered file by searching successively smaller segments of the file until the record is found. Following is a simplified description of this procedure: the highest and lowest records in the file are checked first; if neither is the desired record, then the middle record is checked. If the middle record is not the desired record, then the sought-after record must be located either in the top half of the file (that is, between the highest record and the middle record), or in the lower half of the file (that is, between the lowest record and the middle record). The middle record in the appropriate half is then checked, and the process of performing successive bifurcations continues until the record is found (or until it is determined that no such record exists in the file). Obviously a binary search cannot efficiently be performed if the file is not sorted in ascending or descending order.

The use of a binary search can be illustrated with an example from industry. Consider a small company which maintains a customer file on disk. In the simplest case, each record in the file might contain only three fields, a three-digit customer I.D. number, the customer's name, and the customer's credit rating:

I.D.#	NAME	CREDIT RATING
062	JOHN Q. NOBLE	A1

Figure 6-1. Typical Entry in Customer Credit File

The customer credit file is a cataloged file named CREDIT, in which each record occupies a single sector. The file begins at Sector #100, and is sorted in ascending order on the customer I.D. numbers.

SECTOR #	I.D.#	NAME	CREDIT
100	007	ELLER, CAROL	A-2
101	011	HEAVER, HERBERT	B-1
102	012	.	.
103	013	.	.
104	017	.	.
105	022	.	.
106	025	.	.
107	037	.	.
108	039	.	.
109	052	.	.
110	055	FRACK, ALFRED R.	A-1
111	062	.	.
112	073	.	.
113	101	.	.
114	111	.	.
115	123	RAPPE, VIRGINIA S.	B-2
116	128	SOLO, HAN	C-3

Figure 6-2. Typical Customer Credit File (Sorted in Ascending Order)

As you can see, the file contains 17 records. Suppose, now, that one of the customers, Alfred R. Frack, applies for additional credit. Before granting this credit, the credit manager will want to check Mr. Frack's credit rating. One way to locate Mr. Frack's record is to search sequentially through the file until his customer I.D. (055) is found. In the sample file, this technique involves reading and checking 11 records, or slightly more than one half the total number of records in the file. A faster and more efficient way to find the record is to search the file with a binary search. The procedure is as follows:

1. Begin by checking the first (lowest) record in the file and the last (highest) record, to see if either of them is the desired record. In this case, neither the first record (I.D.#007) nor the last record (I.D.#128) is the desired record.

2. Next, check the middle record in the file. To find the sector address of this record, add the sector address of the last (highest) record in the file (116) to the sector address of the first (lowest) record in the file (100), and take the integer value of the average:

   ```
   M = INT((H+L)/2)
   M = INT((116+100)/2)
   M = 108
   ```

   For the first search, the highest address is 116 (H=116), and the lowest is 100 (L=100). Thus, M=108. The first sector to be accessed is sector 108.

3. Compare the key of this record (I.D. #039) with the desired key (I.D. #055). Since the desired key 055 is greater than the middle key 039, it must be located in the top half of the file (that is, between sectors 108 and 116).

4. Using the middle sector address (108) as the new low sector address, find the middle record in the top half of the file, midway between sector 108 and sector 116. In this case, INT((108+116)/2)=112.

5. Retrieve sector 112 and compare its key (I.D. #073) with the desired key (I.D. #055). Since 073 is larger than 055, the desired record must be in the lower quarter of this half of the file (i.e., between sector 108 and sector 112). Using sector 112 as the new high address, find the sector midway between 108 and 112. INT((108+112)/2)=110. Compare the key of sector 110 (I.D. #055) with the desired key (I.D. #055). Since the keys match, sector 110 contains the desired record, and the search is finished.

1st Search

	Sector Address	Key
	100	007
	101	011
	102	012
	103	013
	104	017
	105	022
	106	025
	107	037
middle →	108	039
	109	052
	110	055
	111	062
	112	073
	113	101
	114	111
	115	123
	116	128

2nd Search

	108	039
	109	052
	110	055
	111	062
middle →	112	073
	113	101
	114	111
	115	123
	116	128

3rd Search

	108	039
middle	109	052
and →	110	055
desired	111	062
record	112	073

Figure 6-3. Binary Search Technique

Although this example presumed an odd number of records in the file, the technique is the same for a file which contains an even number of records. A more serious problem is presented by files in which each record consists of two or more sectors. In such a case, the number of sectors in each record must be taken into account when calculating the record addresses on each search. It is impossible to conduct a binary search if the number of sectors per record is not constant.

In order to conduct a binary search on a file, then, there are three requirements:

1. The file must be sorted.

2. The number of sectors per record must be constant.

3. The limits of the file (i.e., beginning and ending sector addresses) must be known.

For cataloged files, the beginning and ending sector addresses can be obtained under program control with the LIMITS statement.

It may be obvious that the ending sector address of a cataloged file should not be used as the upper limit of the file, unless the file is filled with data. Use of the ending sector address as the upper limit when the file is not full decreases the efficiency of the binary search, since one or more searches may be wasted searching the empty sectors between the end-of-file trailer record and the last sector of the file (or, those unused sectors may contain meaningless data - including old program text - which would cause an error when the DATALOAD DA statement attempts to read it). It is generally safer and more efficient to use the address of the last data record as the upper limit of the file in a binary search since all sectors between the beginning of the file and the last data record are certain to contain valid data. The address of the last data record in a file is computed by subtracting 1 from the address of the end-of- file trailer record. The address of the trailer record can be computed by first executing a LIMITS on the file (with the file name specified), then subtracting 2 from the number of sectors used in the file, and adding this value to the starting sector address of the file. Thus, to determine the address of the trailer record in the file "CREDIT", first execute a LIMITS:

```
20 LIMITS F "CREDIT", A1, A2, A3
```

Since the file name is specified rather than a file number, LIMITS accesses the Catalog Index on the 'F' platter and retrieves the starting and ending sector addresses, and number of sectors used, for CREDIT. Variable A1 contains the starting sector address, variable A2 the ending sector address, and variable A3 contains the number of sectors used. The address of the trailer record then is computed with the following formula:

T = Starting +(Used -2)
T = A1 +(A3-2)

The address of the trailer record is stored in variable 'T'. The sector address of the last data record in the file may now be found merely by subtracting one from the address stored in 'T':

H = T-1

Here the address of the last data record is stored in variable 'H'. This address is used as the upper limit of the file for the first dichotomy in the binary search. The following example program illustrates the binary search described above on the customer credit information file, "CREDIT".

Example 6-26: Performing a Binary Search on a Cataloged Data File

```
10      DIM R$3, A$3, F$26, C$4
20      LIMITS F "CREDIT",A1,A2,A3
25  REM ****** COMPUTE ADDRESS OF LAST DATA RECORD ******
30      T = A1+(A3-2)
40      H = T - 1
50  REM ****** ENTER KEY OF DESIRED RECORD ******
60      INPUT "DESIRED I.D.",R$
70  REM ****** READ & CHECK LOWEST RECORD ******
80      DATA LOAD DA F (A1,S) A$,F$,C$
90      IF A$ = R$ THEN 260
100 REM ****** READ & CHECK HIGHEST RECORD ******
110     DATA LOAD DA F (H,S) A$,F$,C$
120     IF A$ = R$ THEN 260
130 REM ****** COMPUTE MIDDLE SECTOR ADDRESS ******
140     M = INT((A1+H)/2)
150 REM ****** READ & CHECK MIDDLE RECORD ******
160     DATA LOAD DA F (M,S) A$,F$,C$
170     IF A$ = R$ THEN 260
180 REM ****** IS DESIRED KEY HIGHER OR LOWER THAN KEY READ? ******
190     IF R$ < A$ THEN 210
200     A1 = M
201     GOTO 230
210     H = M
220 REM ****** HAVE ALL RECORDS BEEN CHECKED? ******
230     IF H = M+1 THEN 280
240     GOTO 140
250 REM ****** RECORD FOUND - PRINT RECORD ******
260     PRINT A$,F$,C$
265     STOP
270 REM ****** RECORD NOT FOUND - PRINT ERROR MESSAGE ******
280     STOP "RECORD NOT IN FILE"
```

Statement 20 performs a LIMITS on the cataloged file CREDIT; the starting sector address of CREDIT is returned to A1, the ending sector address to A2, and the number of sectors used to A3. Statement 30 calculates the address of the trailer record in CREDIT by subtracting 2 from the number of sectors used (A3), and adding this value to the starting address (A1). The resultant address is stored in T. Statement 40 computes the address of the last data record by subtracting 1 from 'T'. Line 60 is an INPUT statement which requests the key for the desired record. Line 80 loads in the first record of the file; its key is checked against the specified key. If there is no match, the highest record in the file is loaded (line 110), and its key is checked (line 120). If neither the first nor the last record is the desired record, the address of the middle record is computed (line 140), and this record is read and checked. If the middle record does not hold the desired key, the process is repeated on the upper or lower half of the file, depending upon whether the desired key is larger or smaller than the middle record key (lines 190, 200). The process continues either until the desired record is found (in which case it is printed), or until it is determined that no such record exists in the file (in which case an error message is displayed).

6.8 CONCLUSION

Direct addressing statements and commands can be used in conjunction with catalog procedures to develop an efficient and versatile data management system. One technique which might be used in such a system is the binary search technique discussed in the preceding section. A variety of different techniques also are available, and the interested reader is directed to the bibliography in Appendix D for a list of texts which discuss disk file access techniques. The direct addressing statements need not, of course, be regarded as merely supplemental to and supportive of catalog procedures. On the contrary, highly sophisticated and complex data management systems can be constructed in Absolute Sector Addressing Mode exclusively. The bibliography in Appendix D also lists a number of texts which discuss disk management system design concepts and philosophies.

CHAPTER 7
ABSOLUTE SECTOR ADDRESSING STATEMENTS AND COMMANDS

7.1 INTRODUCTION

This chapter contains descriptions of and General Forms for the following Absolute Sector Addressing statements and commands, listed alphabetically for ease of reference:

COPY
DATALOAD BA
DATALOAD DA
DATASAVE BA
DATASAVE DA
LOAD DA (command)
LOAD DA (statement)
SAVE DA

7.2 STATEMENT/COMMAND DISTINCTION AND GENERAL RULES OF SYNTAX

Refer to Chapter 5, Section 5.2, for an explanation of the distinction between BASIC-2 statements and commands.

Refer to Chapter 5, Section 5.3, for a list of the rules of syntax and notation used in the General Forms.

COPY

General Form:

```
COPY platter ⎡file#,       ⎤ [ [ (start-sector, end-sector) ] TO platter ⎡file#,       ⎤ [ (destination-sector) ]
             ⎣disk-address,⎦                                              ⎣disk-address,⎦
```

where:

start-sector	= An expression whose value equals the address of the first sector to be copied.
end-sector	= An expression whose value equals the address of the last sector to be copied.
destination-sector	= An expression whose value equals the starting sector address on the destination platter.

Purpose:

The purpose of the COPY statement is to copy information from one disk platter to another even when both platters are located in separate disk units. In the COPY statement, the value of 'start' represents the address of the first sector to be copied, and the value of 'end' represents the address of the last sector to be copied.

If the 'start' and 'end' parameters are omitted, the entire Catalog Index and Catalog Area (up to the current end) are copied. If the disk is not cataloged, the 'start' and 'end' addresses must be specified. COPY does not delete scratched files. The specified sectors are copied to the specified destination platter starting at 'sector'. If the 'sector' is not specified, it defaults to the 'start' address.

When COPY is executed, approximately 800 bytes of memory must be available for buffering (that is, at least 800 bytes of memory must not be occupied by a BASIC program or variables); otherwise, an ERROR A03 (Memory Overflow) is signalled and the COPY is not performed. The large buffer minimizes the time required for the COPY operation.

Following the COPY, a VERIFY statement can be executed to ensure that the specified information was written correctly on the destination platter.

Examples of valid syntax:

```
100 COPY F TO R
100 COPY R/310, TO F/320,
100 COPY T # A, TO T # B,
100 COPY F (1000,2000) TO R
100 COPY R (100,200) TO R (300)
100 COPY T/D10, TO T/D14,
```

General Form:

```
                         [ file#,       ]
DATALOAD BA platter      [              ] (sector-address [,[return-variable] ] ) alpha-array
                         [ disk-address,]
```

where:

sector address =	An expression or alphanumeric variable whose value specifies the sector address of the record to be read. The value of the expression or alpha variable must be less than or equal to the last (highest) sector address on the disk platter.
return-variable =	A variable which is set to the address of the next sequential sector after the DATALOAD BA statement is processed.

Purpose:

The DATALOAD BA statement is used to load one sector of unformatted data from the disk. The 'BA' parameter specifies Absolute Sector Addressing Mode and block data format, and is not normally used when the referenced file is a cataloged file. The DATALOAD BA statement reads one sector from the specified disk and sequentially stores the entire 256 bytes in the designated alpha-array (or STR() of alpha-array). No check is made for control bytes normally found in System 2200 data records. An error results if the alpha-array is not large enough to hold at least 256 bytes. If the array is larger than 256 bytes, the additional bytes of the array are not affected by the DATALOAD BA operation.

After the statement is executed, the system returns the address of the next consecutive sector, either as a decimal value if a numeric return-variable is specified, or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement or command to provide sequential access to data stored on the disk.

Execution of the DATALOAD BA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

Examples of valid syntax:

```
100 DATALOAD BA F (20) A$( )
100 DATALOAD BA T #2, (B$,B$) B$( )
100 DATALOAD BA F /320, (C,C) J$( )
100 DATALOAD BA T #A, (A,B) STR(A$( ),X,256)
100 DATALOAD BA T/D13, (30) A$( )
```

DATALOAD DA

General Form:

$$\text{DATALOAD DA platter} \begin{bmatrix} \text{file\#,} \\ \text{disk-address,} \end{bmatrix} \text{(sector-address[,[return-variable]]) argument-list}$$

where:

sector-address = An expression or alphanumeric variable whose value specifies the starting sector address of the record to be loaded. The value of the expression or alpha variable must be less than or equal to the last (highest) sector address on the disk platter.

return-variable = A variable which is set to the address of the next available sector after the DATALOAD DA statement is processed.

$$\text{argument list} = \left\{ \begin{matrix} \text{variable} \\ \text{array} \end{matrix} \right\} \left[, \left\{ \begin{matrix} \text{variable} \\ \text{array} \end{matrix} \right\} \right] \ldots$$

Purpose:

DATALOAD DA reads one or more logical records from the disk, starting at the absolute sector address specified. The 'DA' parameter specifies direct addressing mode and generally is not used when the referenced data file is a cataloged file. However, Absolute Sector Addressing can be used with cataloged files and may be useful for certain applications (see Section 9.8). The data to be read must be in standard System 2200 format, including the necessary control information (i.e., the data must have been written onto the disk by a DATASAVE DA or DATASAVE DC statement).

The DATALOAD DA statement reads a logical record from the specified disk and assigns the values read to the variables and/or arrays in the argument list sequentially. An error results if numeric data is assigned to an alpha variable, or vice versa. If data assigned to an alpha variable is shorter than the length of the variable, the value is padded with trailing spaces; if the value is longer, it is truncated. Arrays are filled row by row.

It should be noted that alpha arrays (e.g., A$( )) receive a separate data value for each element of the array. However, the STR( ) of an array receives only a single value.

If the argument list in the DATALOAD DA statement is not filled, another logical record is read. Data in the logical record not used by the DATALOAD DA statement is read but ignored. If the argument list requires more data than is contained in the logical record being read, data is automatically read from the next logical record until the argument list is satisfied. The remainder of the next record is then read but ignored. If an end-of-file (trailer record) is encountered while executing a DATALOAD DA statement, no additional data is read, the next available sector is set to the sector address of the trailer record, and the remaining variables in the argument list remain at their current values. An IF END THEN statement will then cause a valid program transfer.

After the DATALOAD DA statement is executed, the system returns the address of the next sequential logical record, either as a decimal value if a numeric return-variable is specified or as a binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement or command to provide sequential access to data stored on disk.

Execution of the DATALOAD DA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

Examples of valid syntax:

```
100 DATALOAD DA R (A$,L$) X, Y( ), Z$( )
100 DATALOAD DA T #3, (20) A$, B2$( ), M2
100 DATALOAD DA F /320, (D,D) F$( ), J
100 DATALOAD DA R (B$,B$) A,B,S( )
100 DATALOAD DA T #A, (E,D) STR (X$( ),Y,200)
100 DATALOAD DA T /D11, (C$,C$),R,S,T( )
```

DATASAVE BA

General Form:

$$\text{DATASAVE BA platter [\$]} \begin{bmatrix} \text{file\#,} \\ \\ \text{disk-address,} \end{bmatrix} \text{(sector-address[,[return-variable]])} \begin{Bmatrix} \text{alpha-variable} \\ \\ \text{literal} \end{Bmatrix}$$

where:

sector-address=	An expression or alphanumeric variable whose value specifies the sector address at which the record is to be saved. The value of the expression or alpha variable must be less than or equal to the value of the last (highest) sector address on the disk platter.
return-variable=	A variable which is set to the address of the next sequential sector after the DATASAVE BA statement is processed.

Purpose:

The DATASAVE BA statement is used to save data on the disk with no control bytes. The 'BA' parameter specifies Absolute Sector Addressing Mode and should be used with caution when referencing a cataloged data file. 'BA' also specifies block data format; each DATASAVE BA statement writes one sector with no control information. The alpha-variable or literal-string contains the data to be written (trailing spaces are also written). If the data to be written is longer than 256 bytes, only the first 256 bytes are written on disk. If the data is shorter than 256 bytes, the remainder of the sector is filled with unpredictable data.

The DATASAVE BA statement writes data to the specified sector on disk. After the statement is executed, the system returns the address of the next sequential sector, either as a decimal value if a numeric return variable is specified, or as a two-byte binary value if an alphanumeric return variable is specified. This address can be used in a subsequent disk statement to permit sequential storage of data on the disk.

The '$' parameter specifies that a 'read-after-write' verification check be made on all data written to the disk. This verification check not only provides added insurance that data is written accurately on the disk, but also substantially increases the execution time of the DATASAVE BA statement.

Since information written with DATASAVE BA contains no control information, it can be read back only with a DATALOAD BA statement (unless proper control information is included explicitly in the data written).

Execution of the DATASAVE BA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

Examples of valid syntax:

```
100 DATASAVE BA F (L$,L$) A$( )
100 DATASAVE BA R $ #3, (20) B$( )
100 DATASAVE BA F $ /320, (2*I,L) F$( )
100 DATASAVE BA T #2, (Q,Q) HEX (438D9247)
100 DATASAVE BA T /D12, (50) B$( )
```

General Form:

```
                                 ┌ file#,        ┐                                            ┌ END           ┐
DATASAVE DA platter [$]          │               │ (sector-address[,[return-variable]] )      <               >
                                 └ disk-address, ┘                                            └ argument-list ┘
```

where:

sector-address = An expression or alphanumeric variable whose value specifies the starting sector address of the record to be saved. The value of the expression or alpha variable must be less than or equal to the last (highest) sector address on the disk platter.

return-variable = A variable which is set to the address of the next available sector after the DATASAVE DA statement is processed.

```
                  ┌ variable   ┐   ┌┌ variable   ┐┐
argument list  =  < literal    > , [< literal    >] • • •
                  │ expression │   ││ expression ││
                  └ array      ┘   └└ array      ┘┘
```

Purpose:

The DATASAVE DA statement is used to save data on the disk in Absolute Sector Addressing Mode. The 'DA' parameter indicates a direct addressing operation; the statement therefore generally is not used when the referenced data file is a cataloged file, since there is a risk the user may unintentionally destroy part of the catalog information. However, direct addressing statements can be used with cataloged files for certain applications (see Section 6.8). The 'END' parameter in a DATASAVE DA statement should never be used for records stored in a cataloged file. There are two important considerations which must be kept in mind when writing a record into a cataloged file with DATASAVE DA. First, the system provides no automatic boundary checking; hence, records can be written past the end of one file and into the beginning of the next without system detection. Second, the "number of sectors used" is not updated in the Catalog Index when a trailer record is written with DATASAVE DA END. Therefore, DSKIP END cannot be used to skip to the end of the file.

The 'DA' parameter specifies that the data in the argument list is to be written in standard System 2200 format, including the necessary control information. Each DATASAVE DA statement writes a logical record consisting of one or more sectors. The DATASAVE DA statement causes the values of variables, expressions, and array elements to be written sequentially onto the specified disk. Arrays are written row by row. It should be noted that each element of an array is written as a separate data value. However, the STR( ) of an alpha-array represents a single data value. Alphanumeric values must be $\leq$ 124 bytes in length.

NOTE:

Each numeric value in the 'argument list' requires 9 bytes on disk; each alphanumeric variable requires the maximum number of characters for which the variable is dimensioned plus 1. Each 256-byte sector also requires three bytes of control information.

If the 'END' parameter is used, a data trailer record is written for the file. This record can be used to test for the end of a file during processing with an IF END THEN statement.

The DATASAVE DA statement writes the data from the argument list onto the disk beginning at the specified sector address. After the statement is executed, the system returns the address of the next available sector, either as a decimal value if a numeric return variable is specified or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in subsequent disk statements to provide sequential access to data on the disk.

The '$' parameter specifies that a read-after-write verification test be made on all data written to the disk. This verification check not only provides added insurance that data is written accurately on the disk, but also substantially increases the execution time of the DATASAVE DA statement.

Execution of the DATASAVE DA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

Examples of valid syntax:

```
DATASAVE DA F (20) X, Y( ), Z$( )
DATASAVE DA R $ /320, (C,C) F$( ), A( )
DATASAVE DA T $ #2, (B$,B$) M$( ), "J.CARTER"
DATASAVE DA F (2*M+1,L) J( ), K1
DATASAVE DA T (Q,Q) END
DATASAVE DA T #A, (A,B) END
DATASAVE DA T $/D13, (A,A)P$( ),R( )
```

General Form:

```
              ⎡file#,       ⎤
LOAD DA platter⎢             ⎥ (sector-address[,[return-variable]] )
              ⎣disk-address,⎦
```

where:

sector-address = An expression or alphanumeric variable whose value specifies the starting sector address of the program to be loaded. The value of the expression or alpha variable must be the address of the program header record, and must be less than or equal to the last (highest) sector address on the disk platter.

return-variable = A variable which is set to the address of the next available sector after the LOAD DA command is processed.

Purpose:

The LOAD DA command is used to load BASIC programs or program segments from the disk in Absolute Sector Addressing Mode. When the LOAD DA command is executed, the program which begins at the specified sector address is read and appended to the current program in memory. (Note that the sector address must be the address of a program header record.) The LOAD DA command can be used to add program text to a program currently in memory or, if entered after a CLEAR command, to load a new program from the disk.

After the LOAD DA command is executed, the system returns the address of the next available sector, either as a decimal value if a numeric return variable is specified or as a two-byte binary value if an alphanumeric return variable is specified. This address can be used in a subsequent disk statement or command to permit sequential access to programs on the disk.

Execution of the LOAD DA command does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the command).

LOAD DA can also be used as a program statement to chain programs or subroutines (see LOAD DA statement).

Examples of valid syntax:

```
LOAD DA R (24)
LOAD DA F (A$,B$)
LOAD DA R /320, (L$,L$)
LOAD DA T #2, (A,B)
LOAD DA R (24,L$)
LOAD DA R (A$,B)
LOAD DA T #A, (C,D)
LOAD DA T /D10, (100)
LOAD DA T (100)
```

LOAD DA (Statement)

General Form:

```
              ┌ file#,       ┐
LOAD DA platter│              │ (sector-address[,[ return-variable]])[line-number 1[,[line-number 2]]] [BEG begin-line-number]
              └ disk-address ┘
```

where:

sector-address	= An expression or alphanumeric variable whose value specifies the starting sector address of the program which is to be loaded. The value of the expression or alpha variable must be the address of the program header record, and must be less than or equal to the last (highest) sector address on the disk platter.
line-number-1	= The line number of the first line to be deleted from the program currently in memory before loading the new program. After loading, execution continues automatically starting at this line number. An error results if there is no line with this number in the new program.
line-number-2	= The number of the last text line to be deleted from the program currently in memory before loading the new program.
begin-line-number	= The line number of the program where execution is to begin after the program is loaded into memory.
return-variable	= A variable which is set to the address of the next available sector after the LOAD DA statement is processed.

Note: The return-variable must be a common variable.

Purpose:

The LOAD DA statement is used to load programs from a specified location on the disk. (Note that the sector address specified must be the address of the program header record.) The 'DA' specifies direct addressing; therefore, the LOAD DA statement is not generally used to load cataloged programs from the disk. LOAD DA is a BASIC program statement which, in effect, produces an automatic combination of the following:

STOP	(Stops current program execution.)
CLEAR P	(Clears program text from memory, beginning at 'line-1' (if specified) and ending at 'line-2' (if specified); if no line number is specified, clear all program text from memory.) The Internal Table and subroutine stacks in memory are cleared.
CLEAR N	(Clears all non-common variables from memory.)
LOAD DA	(Loads new program or program segment from disk.)
RUN	(Runs new program, beginning at 'begin' if specified). If the 'BEG' parameter is not specified, program execution begins at 'line-1', or at the lowest program line in memory if 'line-1' is not specified.

The two line number parameters may be used to cause the system to clear a specified portion of resident program text prior to loading in the new program. If both line numbers are specified, all program lines between and including the two specified lines are cleared prior to loading the new program. If only 'line-1' is specified, the remainder of the resident program is deleted starting with that line number. If only 'line-2' is specified, all program lines up to and including 'line-2' are deleted. If no line numbers are specified, the entire resident program is deleted. In every case, all non-common variables are cleared. LOAD DA permits segmented programs to be run automatically without normal user intervention, with common variables passed between program segments. If included on a multi-statement line, LOAD DA must be the last executable statement on the line.

After the program is loaded, the system returns the address of the next sequential sector either as a decimal value, if a numeric return-variable is specified, or as a two-byte binary value, if an alphanumeric return-variable is specified. This address can be used in a subsequent statement to permit sequential access to programs on the disk.

Execution of the LOAD DA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

In Immediate Mode, LOAD DA is interpreted as a command (see LOAD DA command).

Examples of valid syntax:

```
 100 LOAD DA F (40)
  50 LOAD DA R /320, (L$,L$) 310,450
 530 LOAD DA T #2, (N$,L$) 570
 700 LOAD DA F /320, (L,L)
1020 LOAD DA F (2*I+1,L$) 400
2000 LOAD DA T #B, (C,D)
3000 LOAD DAF (L$,L$) 310,450 BEG 400
 500 LOAD DA T/D14, (J,K)
 570 LOAD DA T #15, (100)
```

NOTE:

When a LOAD DA statement is executed, the system stacks are cleared of all subroutine and loop information.

SAVE DA

General Form:

```
         [<S> ]            [ file#,        ][!]
SAVE DA  [    ] platter [$][               ][ ]  (sector-address[[,return-variable]]) [start-line-number] [,[end-line-number]]
         [<SR>]            [ disk-address, ][P]
```

where:

<S>	= A parameter specifying that unnecessary spaces (not including spaces in character strings enclosed in quotes, REM or % statements) will be deleted from a program as it is saved.
<SR>	= A parameter specifying that both spaces and remarks (REM statement lines) are deleted from the program as it is saved.
!	= Protect (scramble) the file to be saved.
P	= Set the protection bit on the file to be saved.
sector-address	= An expression or alphanumeric variable whose value specifies the starting sector address of the program to be saved. The value of the expression or alpha variable must be less than or equal to the last (highest) sector address of the disk platter.
return-variable	= A variable which is set to the address of the next available sector after the SAVE DA command is processed.
start-line-number	= The number of the first program line to be saved.
end-line-number	= The number of the last program line to be saved.

Purpose:

The SAVE DA command is used to save programs on the disk beginning at a specified location. Because the 'DA' specifies Absolute Sector Addressing Mode, this command should not be used if the program is to be saved under catalog procedures. SAVE DA causes BASIC programs (or portions of BASIC programs) to be recorded on the designated platter beginning at the specified sector address. The program cannot be named and can be loaded back into memory only with a LOAD DA statement or command.

After each program is saved, the system returns the address of the next available sector, either as a decimal value if a numeric return variable is specified, or as a two-byte binary value if an alphanumeric return variable is specified. This address can be used in a subsequent disk command to permit the sequential storage of programs on disk.

The 'start-line-number' and 'end-line-number' parameters specify the first and last lines, respectively, of the program in memory which is to be saved. Both these parameters are optional; if only 'start' is included, all program lines in memory beginning with that line through the end of the program are saved. If only the 'end' line number is specified, all program text from the beginning of the program through the specified line are saved. If no line numbers are specified, all program text in memory is saved.

Execution of SAVE DA does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the command).

The '\$' specifies that a read-after-write verification check be performed on all information written to the disk. This verification check not only provides added insurance that the program is recorded accurately, but also substantially increases the execution time of SAVE DA.

When saving a program on disk, nonessential spaces and/or remarks can be deleted from a program as it is saved by the '< >' parameter. An <S> signifies that any unnecessary spaces (except spaces in character strings enclosed in quotes and % or REM statements) are to be deleted.

> **NOTE:**
>
> The system automatically inserts spaces after line numbers, statement separators (colons), and most BASIC words for readability when displaying a program line. These extra spaces are not in the program in memory.

An <SR> in the SAVE DA statement means delete both spaces and remarks (REM statement lines). The <SR> causes all REM statements to be deleted from the program. If certain REM's are to be saved, the REM statements can be changed to image statements (%). Image statements are not disturbed by <SR>.

The 'P' parameter permits a program to be protected against accidental modification. After a program that has been saved via SAVE P is loaded, the program in memory cannot be modified (except by overlaying). The operator cannot, then, inadvertently modify the program. SAVE P also prevents the program from being listed or resaved.

The '!' parameter performs the same function as 'P' but also provides a secure means of preventing program examination. The entire program is scrambled when recorded on disk, and cannot be examined via DATALOAD BA.

The error message display for protected programs is modified to supress the display of the program text. Only the line number, statement separators and error code are displayed.

Example:

```
100 X = 1: Y = 2: Z = 1/0
```

RUN (EXEC)

Output: (unprotected program)

```
100 X = 1: Y = 2: Z = 1/0
                        ↑ERR C62
```

Output: (protected program)

```
100:: ERR C62
```

NOTE:

In order to save any program on disk after a protected program has been loaded, the user must enter a CLEAR command with no parameters, or Master Initialize the system (i.e., turn main power switch OFF, then ON).

Examples of valid syntax:

```
    SAVE DA T (3)
10  SAVE DA R $ /320, P (L,L)
10  SAVE DA R #2, (A$,A$) 200
    SAVE DA T #2, P (A$,A$)
    SAVE DA F!(2+X,L)
    SAVE DA<SR> F(A$,A$)
    SAVE DA T/D10, (1000)
```

CHAPTER 8
THE DISK MULTIPLEXER (MODEL 2230MXA-1/MXB-1)

8.1 INTRODUCTION

When more than one CPU is configured to have access to a common disk data base, a multiplexed disk environment exists. Multiplexing adds an important dimension to disk ownership. A single disk unit can be apportioned among several offices or departments. Each will have access to the disk data base while retaining its own system in a convenient location. The participating systems may share a common data base on disk, or each system may have a specified portion of the disk reserved for its own use. In either case, the disk receives maximum utilization. Each user is provided with a random-access mass-storage capability, and the costs incurred by any one user are reduced. The disk operations from multiple inquiring systems are interleaved, and disk time is allocated among the inquiring systems in a manner which provides all systems with virtually concurrent access to the disk.

With the Model 2230MXA-1/MXB-1 up to four independent CPU's can be multiplexed to the same disk unit. The Model 2230MX is a "daisy-chain" multiplexer which consists solely of a series of special multiplexer controller boards. The 2230MXA-1 "master" board, installed in the primary CPU, controls all access to the disk unit. The 2230MXB-1 "slave" boards are installed in participating CPU's, and the slave CPU's are connected together to form a chain. *Only* the system with the master board connects directly to the disk drive.

NOTE:

The following disk drives can be multiplexed with the Model 2230 MXA-1/MXB-1: 2260BC, 2270, 2270A.

The Model 2280 Disk Drive can be multiplexed with the 2280 Disk Multiplexer (Refer to Appendix F).

8.2 THE MODEL 2230MX MULTIPLEXER

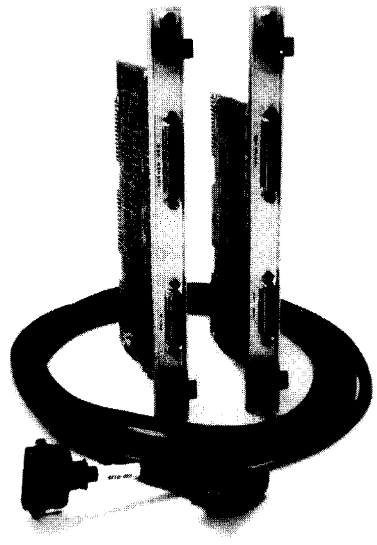


Figure 8-1. Model 2230MXA-1 Master Board and 2230MXB-1 Slave Boards

The Model 2230MX Multiplexer is a "daisy chain" multiplexer consisting of a single 2230MXA-1 master controller board, and one to three 2230MXB-1 slave controller boards. The 2230MXA-1 master board and 2230MXB-1 slave boards are purchased separately. The number of slave boards required is determined by the size of the total installation: a master board and one slave board permit two stations to share the disk, a master board and two slave boards permit three stations to share the disk, a master board and three slave boards permit four stations to share the disk.

The master board has a 50-pin input connector, labeled "MUX OUTPUT", and a 36-pin connector labeled "DISK". Each slave board has a 36-pin input connector labeled "MUX IN", and a 50-pin output connector labeled "MUX OUT".

The PCSIIA and 2200 workstation have a built in slave board with a connector labeled "DISK". A T-connector must be used unless the system is at the end of the multiplexer chain.

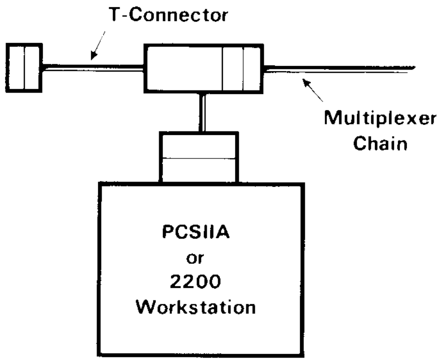


Figure 8-2. T-connector with Multiplexed System

The connector cables correspondingly have two plugs, one of 36 pins and one of 50 pins. The systems are connected together by running cables from the MUX OUT jack of one CPU to the MUX IN jack of the next consecutive CPU to form a chain. At the beginning of the chain is the master system (the CPU with the 2230MXA-1 master board). The disk connector cable plugs into the DISK jack on the master board to complete the chain. (See Figure 8-4.) The master system is the only system which connects directly to the disk unit.

In addition to the standard 12-foot (3.7m) connector cable shipped with each slave board, longer extension cables are available in lengths of 50, 100, and 200 feet (15.5, 31, and 61 meters). The extension cable part numbers are listed below:

Cable Length	Part #
50 ft (15.5m)	120-2225-50
100 ft (31m)	120-2225-100
200 ft (61m)	120-2225-200

These cables are "extension cables" in a literal sense since they serve as extensions for the standard connector cables; an extension cable cannot be used by itself to connect two systems. Each extension cable has two 36-pin plugs, one male and one female. The male plug is inserted in the MUX IN jack of a slave board, while the female plug must be connected to the 36-pin male plug on a standard connector cable. The 50-pin plug on the other end of the standard cable is then inserted in the MUX OUT jack of a second board. Because the extension cable is combined with the standard cable in this manner, the total length of the cable between two units is always equal to the extension cable length plus 12 feet. (See Figure 8-3.)

In special cases, it is possible to connect two or more extension cables together to create an extension longer than 200 feet. In every case, however, the maximum permissable distance between two systems is 512 feet, and the maximum distance between the first and last systems in the chain is 536 feet. The cable connecting the disk unit to the master CPU is approximately ten feet (3 meters) in length, and cannot be extended.

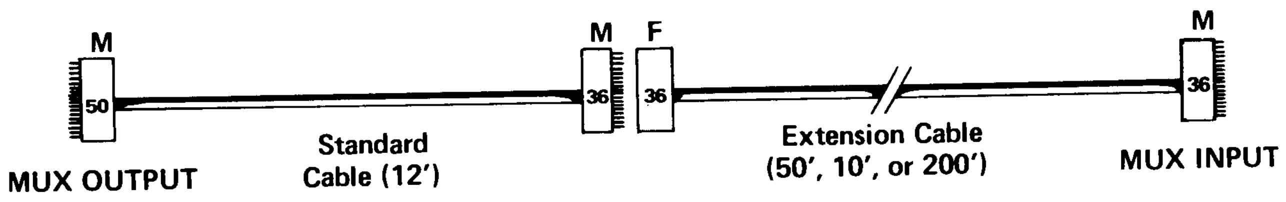


Figure 8-3. Connecting Extension Cable with Standard 12-foot Cable

8.3 INSTALLING THE MODEL 2230MX

Unpacking and Inspection

Carefully unpack your equipment and inspect it for damage. If a unit is damaged, notify the shipping agency at once. Be certain that you have one 2230MXA-1 master board, and the expected number of 2230MXB-1 slave boards. Note: the DISK jack on the 2200 Workstation is equivalent to a slave multiplexer controller board jack.

Installation Procedure

NOTE:

If a connector cable is to be routed through a conduit or any tight space requiring removal of a plug, it is important that the plug be disconnected and reconnected by a qualified Wang Service Representative. Reconnection of the plug is a delicate job which, if done improperly, can impede or prevent data transmission along the line. Contact your Wang Field Service Office to perform all installation service.

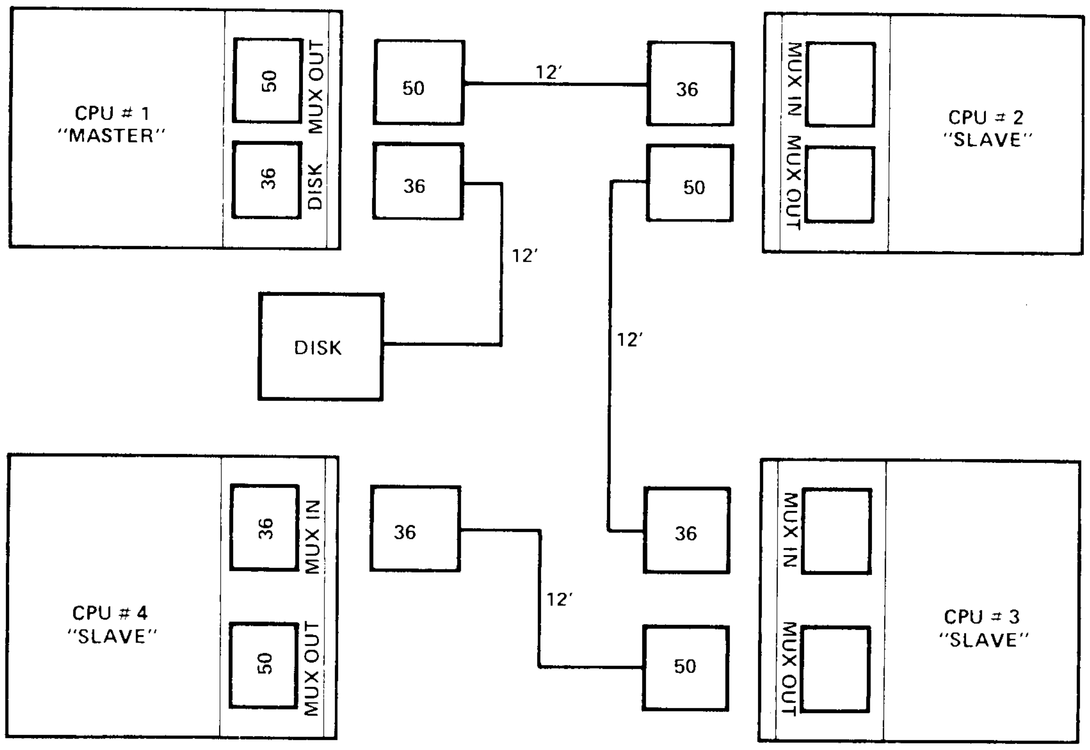


Figure 8-4. Typical System Configuration: Model 2230 MX Multiplexer, Disk Unit, and Four Attached CPU's.

1. Install the 2230MXA-1 master controller board in CPU #1 (the system nearest the disk). Install 2230MXB-1 slave boards in the remaining systems. In systems which already have a disk controller board, the multiplexer board replaces the disk board.

2. Plug the disk I/O cable into the jack marked "DISK" on the 2230MXA-1 master board.

3. Insert the 50-pin connector cable plug in the jack labelled "MUX OUTPUT" on the master board. If no extension cable is used, insert the 36-pin plug on the other end of the cable into the MUX IN jack in the slave board in CPU #2 (or into the T-connector cable if the next system is a PCS-IIA or 2200 workstation). If an extension cable is used, plug the standard cable into the extension cable, and plug the extension cable into the MUX INPUT jack. Repeat this procedure for all attached systems.

4. Be sure that all attached systems are properly set up and ready for operation.

5. Plug all power cords into grounded (three-hole) wall sockets.

NOTE:

When routing the multiplexer connector cables between participating systems, take care to avoid exposing a cable to intense electric or magnetic fields, or sources of electronic noise, since they may interfere with data transmission over the cable. In general, you should try to keep the connector cable away from electrical trunk lines, fluorescent lights, and electrical office equipment (such as electric typewriters and tape recorders). If you have a specific question about routing a cable, contact your Wang Service Representative.

Power-On Procedure

1. Switch ON the power switches on all system peripherals, including the disk unit.

2. Switch ON the Main Power Switches on all system CPU's.

3. The POWER light should illuminate on the disk unit. The CRT display at each station looks like this:

```
MOUNT SYSTEM PLATTER
PRESS RESET
```

NOTE:

When several systems are multiplexed to the same disk with the 2230MX Multiplexer, the master CPU (the CPU with the 2230MXA-1 master board) must be powered ON before any other system can access the disk. However, one or more of the slave CPU's may be OFF without disturbing the operation of the other CPU's. Powering on or off while the disk is in use may cause disk errors to occur.

4. Touch RESET on the keyboard of the master system to initialize the controller. The disk may now be accessed via the multiplexer from any attached system. Turn to Section 8.4 for an explanation of how the multiplexer operates, and a discussion of some programming considerations.

NOTE:

If you experience difficulty in maintaining valid data transmission between the disk and one or more systems, the problem may lie in the connector plugs. A coating sometimes forms on the pins of a plug during extended periods of disuse. To remove this coating, which may inhibit transmission, simply insert and remove the plug in a jack several times, or cut a piece from an ink-type eraser small enough to fit between the pins, and use it to clean the surfaces of the pins. (Transmission problems also can be created by electrical and magnetic interference in the cables.)

8.4 MULTIPLEXER OPERATION

The disk multiplexer controls all communication between participating systems and the disk unit. The multiplexer automatically "polls" each system, beginning with system (or "station") #1, until it finds a system which is attempting to access the disk. At that point, the multiplexer permits the inquiring station to execute one disk statement or command. Following execution of the statement or command, the multiplexer resumes its polling until it encounters another system trying to access the disk. The multiplexer does not monitor the amount of time required to execute each statement, nor does it limit the number of sectors transferred by a statement. A single statement may read or write only one sector, but it is equally possible to carry out multi-sector transfers with one statement. (A MOVE or COPY statement, for example, might transfer an entire disk platter to a second platter.) It is recommended, however, that major file maintenance operations be executed only by a station in Hog Mode (see below). In any case, the system which is executing the statement retains use of the disk until statement execution is completed. Control is then transferred to the next inquiring station. The Model 2230MX provides no external indication of which system has access to the disk.

In normal operation, the multiplexer imposes no special demands or conditions upon the programmer. The disk is simply addressed as usual with the appropriate disk statements and commands. If no other systems are accessing the disk, the total execution time of a multi-statement disk operation is not noticeably affected by the Multiplexer. If more than one multi-statement disk operation is being carried on at once, however, the time required for each operation is roughly equal to the total time required to execute all operations, since one statement from each system is executed on each pass by the multiplexer.

Although in general all systems attached to the multiplexer gain access to the disk on a statement-by-statement basis, there are cases in which it is desirable to give one system a period of exclusive and uninterrupted access to the disk. During certain critical file maintenance or update procedures, for example, it is important that other systems be prevented from accidentally interfering in the routine, since they might unknowingly overwrite valuable data or pointers, or otherwise confuse the situation. Because operators on remote stations have no way of knowing that critical maintenance procedures are being carried out at any given time, it is necessary to prevent them from unknowingly interrupting a routine by locking them out. A system which monopolizes the disk in this way is said, somewhat picturesquely, to be "hogging" the disk. Note that *every* disk platter in the disk unit is hogged when the disk unit is hogged.

Whenever a CPU is granted access to a disk platter, it automatically gains control of all platters associated with that disk drive. A station that has the disk in hog mode can execute any number of disk statements or commands while maintaining exclusive control of the disk, preventing other stations from executing any operation on the hogged disk drive.

8.5 HOG MODE

The disk drive may be hogged by either of two methods: $GIO hog, or address hog. $GIO hog consists of a series of microcommands directed from the CPU to the MXA-1 controller board. In hog address, the disk is accessed using special disk addresses, called "hog mode addresses". The disk remains hogged until a disk statement accesses the disk with the normal disk address.

The $GIO hog is recommended since it instructs the multiplexer to hog or unhog the disk without actually performing a disk operation. Furthermore, with $GIO hog the program need not be concerned with two sets of disk addresses since the normal disk addresses are always used with this form of disk hog; unhogging is done with the $GIO DISK RELEASE statement.

$GIO Hog

The general form of $GIO hog is as follows:

a) to hog the disk:

$$\text{\$GIO DISK HOG} \left\{ \begin{array}{c} \text{file number} \\ \text{or} \\ \text{disk device address} \end{array} \right\} \text{(4480)}$$

b) to release the disk:

$$\text{\$GIO DISK RELEASE} \left\{ \begin{array}{c} \text{file number} \\ \text{or} \\ \text{disk device address} \end{array} \right\} \text{(4400)}$$

File numbers are values which are assigned within programs to replace disk device addresses. For example, SELECT #1/B10 assigns #1 to disk device address B10. Disk device addresses are not programmer selectable, but are preset within each disk controller board.

NOTE TO MVP/LVP USERS:

The recommended statements for HOG MODE on the MVP or LVP (and the VP, Release 1.8 or later) are as follows:

a) to hog the disk:

$OPEN disk device address

b) to release the disk:

$CLOSE disk device address

The user should refer to the BASIC-2 Language Reference Manual for an in-depth discussion of $OPEN and $CLOSE.

Example 8-1: Entering and Leaving Hog Mode Using $GIO Hog

```
110    REM OPEN FILE IN NON-HOG MODE
120    SELECT #1/B20
130    DATA LOAD DC OPEN T#1, "DATAFILE"
.
.      (processing)
.
270    DBACKSPACE #1, BEG
280    DSKIP #1, N S : REM SKIP N SECTORS
290    REM UPDATE RECORD IN HOG MODE
300    $GIO DISK HOG #1 (4480):REM ENTER HOG MODE
310    DATA LOAD DC #1, A,B,C :REM READ RECORD
320    DBACKSPACE #1, 1 S
330    DATA SAVE DC #1, A, B+K, C:REM UPDATE
340    $GIO DISK RELEASE #1 (4400):REM LEAVE HOG MODE
```

This example illustrates a typical update routine in which hog mode is activated temporarily during the actual updating (from the time the record is read until its updated version is written.) The file is opened with the disk drive in non-hog mode (line 130). Lines 270 and 280 locate the desired record also while in non-hog mode. Hog mode is entered upon execution of line 300. (The multiplexer ceases its polling of the stations upon entering hog mode. This station maintains exclusive access to the entire disk drive until executing line 340, when hog mode is left. (The hogging station also loses control of the disk drive if RESET is keyed on the station's keyboard.)

Address Hog

When using address hog, a special disk address, called a "hog mode address", must be used for all disk statements. When, during normal mode operation, the multiplexer finds a station waiting to execute a disk statement with a hog mode address, it gives that station hog mode control of the disk drive, and normal station-polling ceases. The hogging station maintains control of the disk drive until it executes a disk statement with a non-hog mode address (or RESET is keyed on the station's keyboard). As soon as a hogging station completes execution of a disk statement with a non-hog mode address, hog mode is released, and the normal mode station-polling resumes.

For any multiplexed disk device, the hog mode address can be calculated by adding HEX(80) to the device address. Sample non-hog and hog mode addresses are:

NORMAL (NON-HOG) ADDRESS	HOG MODE ADDRESS
310	390
B10	B90
320	3A0
B20	BA0
330	3B0
B30	BB0

The hog-mode addresses refer to the same disks as do their non-hog versions. Thus, if the disk drive normally addresses as 320 is a multiplexed disk, then 3A0 refers to this same disk. The only difference is that when a disk statement is executed at address 3A0, it signifies to the multiplexer that the station executing the disk statement wishes to hog the disk drive.

Example 8-2: Entering and Leaving Hog Mode Using Address Hog

```
290     REM UPDATE RECORD IN HOG MODE
300     SELECT #1/BA0 :REM HOG MODE ADDRESS
310     DATA LOAD DC #1, A,B,C :REM ENTER HOG MODE AND READ RECORD
320     DBACKSPACE #1, 1 S
330     SELECT #1/B20 :REM NON-HOG ADDRESS
340     DATA SAVE DC #1, A, B+K, C:REM UPDATE, THEN LEAVE HOG MODE
 .
 .
 .
```

In the above example line 300 substitutes the hog mode address, BAO, for its non-hog version, B20, in the device table. Note that this does not affect the file parameters, and that, of itself, this does not cause the disk drive to be hogged. Line 310 loads the record, and, since a hog mode address is in file number #1, activates hog mode for the disk drive. After line 310 is executed, the multiplexer ceases its polling of the stations. This stations maintains exclusive access to the entire disk drive, until it executes a disk statement at a non-hog address. Line 320 backspaces one sector. This disk statement takes place at a hog mode address (in file number #1), so hog mode is maintained. Line

330 selects a non-hog address in preparation for leaving hog mode after the next statement. Line 340 updates the record, and, since file number #1 now contains a non-hog address, it returns the disk drive to normal mode after execution is complete.

The following points should be noted in regard to the operation of hog and non-hog mode:

1. When a multiplexed disk drive is hogged, the entire disk unit (all platters) is hogged.
2. Only the stations which activates hog mode can deactivate it.
3. If a station attempts to execute a disk statement while another stations is hogging the disk drive, the station simply waits, with the processing light on, until hog mode is released.
4. Hog mode is deactivated if RESET is keyed at the hogging station.

APPENDIX A
DISK ERROR CODES

ERR D80

Error:	File Not Open
Cause:	The file was not opened.
Recovery:	Open the file before attempting to read from it or write to it.

ERR D81

Error:	File Full
Cause:	The file is full; no more information may be written into the file.
Recovery:	Correct the program, or use MOVE to move the file to another platter and reserve additional space for it.

ERR D82

Error:	File not in Catalog
Cause:	A non-existing file name was specified, or an attempt was made to load a data file as a program file.
Recovery:	Make sure the correct file name is being used, the proper disk is mounted, and that the proper disk drive is being accessed.

ERR D83

Error:	File Already Cataloged
Cause:	An attempt was made to catalog a file with a name that already exists in the Catalog Index.
Recovery:	Use a different name, or catalog the file or a different platter.

ERR D84

Error:	File Not Scratched
Cause:	An attempt was made to rename, or write over a file that has not been scratching.
Recovery:	Scratch the file before renaming it.

ERR D85

Error:	Catalog Index Full
Cause:	There is no more room in the Catalog Index for a new name.
Recovery:	Scratch any unwanted files and compress the catalog using a MOVE statement, or mount a new disk platter and create a new catalog.

ERR D86

Error:	Catalog End Error
Cause:	The end of the catalog area is defined to fall within the catalog index, or an attempt has been made to move the end of the catalog area to fall within the area already occupied by cataloged files (with MOVE END), or there is not room left in the Catalog Area to store more information.
Recovery:	Correct the SCRATCH DISK or MOVE END statement, or increase the size of the Catalog Area with MOVE END, or scratch unwanted files and compress the catalog with MOVE, or open a new catalog on a separate platter.

ERR D87

Error:	No End of File
Cause:	No end-of-file record was recorded in the file (with DATASAVE DC END or DATASAVE DA END), and therefore none could be found by the DSKIP END statement.
Recovery:	Correct the file by writing an end-of-file trailer after the last data record.

ERR D88

Error:	Wrong Record Type
Cause:	A program record was encountered when a data record was expected, or vice versa.
Recovery:	Correct program. Be sure the proper platter is mounted and be sure the proper drive is being accessed.

ERR D89

Error:	Sector Address Beyond End of File
Cause:	The sector address being accessed by the DATALOAD DC or DATASAVE DC operation is beyond the end-of-file. This error can be caused by a bad disk platter.
Recovery:	Run the program again. If error persists, use a different platter or reformat the platter. If error still exists, contact your Wang Service Representative.

ERR I90

Error:	Disk Hardware Error
Cause:	The disk did not recognize or properly respond to the System at the beginning of a read or write operation (the read or write has not been performed).
Recovery:	Run the program again. If error persists, contact your Wang Service Representative.

ERR I91

Error:	Disk Hardware Error
Cause:	A disk hardware error occurred, e.g., the disk is not in file-ready position. This could occur, for example, if the disk is in LOAD mode or power is not turned on.
Recovery:	Ensure disk is turned on and properly set up for operation. Set the disk into LOAD mode and then back into RUN mode with the RUN/LOAD selection switch. The check light should then go out. If error persits, call your Wang Service Representative. (Note: disk must never be left in LOAD mode when turned on.)

ERR I92

Error:	Time-out Error
Cause:	The disk did not respond to the sytem during a read or write operation in the proper amount of time (time-out).
Recovery:	Run program again. If error persists, reinitialize disk - if error still occurs, contact your Wang Service Representative.

ERR I93

Error:	Disk Format Error
Cause:	A disk format error was detected during a disk read or write. The disk is not properly formatted. The error can be either in the disk platter or the disk hardware.
Recovery:	Format the disk again; if error persists, call your Wang Service Representative.

ERR I94

Error:	Format Key Engaged
Cause:	The disk format key is engaged (the key should be engaged only when formatting a disk).
Recovery:	Turn off the format key.

ERR 195

Error:	Seek Error, or Platter Protected
Cause:	A disk-seek error occurred; the specified sector could not be found on the disk. This error may indicate a bad format, or it may result from an attempt to write to a protected platter.
Recovery:	Run program again. If error persists, reinitialize (reformat) the disk. If error still occurs, call your Wang Service Representative.

ERR 196

Error:	Cyclic Read Error
Cause:	A cyclic redundancy check (CRC) error occurred during a disk read operation; the sector being addressed has never been written to or was incorrectly written.
Recovery:	If not formatted, format the disk. If the disk was formatted, rewrite the bad sector. If error persists, use a different disk platter. If error persists on a fixed platter, call your Wang Service Representative.

ERR 197

Error:	LRC Error
Cause:	A disk longitudinal redundancy check (LRC) error occured when reading a sector. This usually indicates a data transmission error occured when the sector was read or written.
Recovery:	If error persists, rewrite the sector. If the error still persists, call your Wang Service Representative.

ERR 198

Error:	Illegal Sector Address or Platter Not Mounted
Cause:	The disk sector being addressed is not on the disk or the disk platter is not mounted. (Maximum legal sector address depends upon the model of disk used.)
Recovery:	Correct the program statement in error, or mount a platter in the specified drive.

ERR 199

Error:	Read After Write Error
Cause:	The comparison of read after write to a disk sector failed, indicating that the information was not written properly. This error usually indicates a bad disk platter.
Recovery:	Write the information again. If error persists, try a new platter; if error still persists, call your Wang Service Representative.

APPENDIX B
COMPARISON OF BASIC AND BASIC-2 DISK STATEMENTS SYNTAX

Those Wang CPU's which recognized BASIC-2 Syntax also recognize and execute Wang BASIC syntax so programs can be written using instructions from either or both of these dialects. However, when programs are to be shared with a CPU which recognizes BASIC syntax only, the programmer must use the Wang BASIC instruction set alone. The syntax and processing differences in the disk instructions of the two Wang BASIC dialects are presented in this appendix.

STATEMENTS

BASIC Syntax

$$\text{COPY} \begin{bmatrix}\text{\#f,}\\ \text{/xyy,}\end{bmatrix} \begin{Bmatrix}\text{FR}\\ \text{RF}\end{Bmatrix} \text{(expr 1, expr 2)}$$

$$\text{DATALOAD BA} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\text{\#f,}\\ \text{/xyy,}\end{bmatrix} \text{(expr 1, return var) alpha-array desig}$$

$$\text{DATALOAD DA} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\text{\#f,}\\ \text{/xyy,}\end{bmatrix} \text{(expr 1, return var) arg-list}$$

DATALOAD DC [#f,] arg-list

$$\text{DATALOAD DC OPEN} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \text{[\#f,]} \begin{Bmatrix}\text{TEMP, expr 1, expr 2}\\ \text{file name}\end{Bmatrix}$$

$$\text{DATASAVE BA} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\text{\#f,}\\ \text{/xyy,}\end{bmatrix} \text{(expr 1, return var) alpha array desig}$$

$$\text{DATASAVE DA} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \ [\$] \begin{bmatrix}\text{\#f,}\\ \\ \text{/xyy,}\end{bmatrix} \ \text{(expr 1, return var)} \begin{Bmatrix}\text{END}\\ \text{expression}\\ \text{arg-list}\\ \text{literal}\end{Bmatrix}$$

$$\text{DATASAVE DC [\$] [\#f,]} \begin{Bmatrix}\text{END}\\ \text{arg-list}\\ \text{expression}\\ \text{literal}\end{Bmatrix}$$

$$\text{DATASAVE DC CLOSE} \begin{bmatrix}\text{\#f,}\\ \\ \text{ALL}\end{bmatrix}$$

$$\text{DATASAVE DC OPEN} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \ \text{[\$] [\#f,]} \begin{Bmatrix}\begin{Bmatrix}\text{old name}\\ \text{space}\end{Bmatrix}\text{, new name}\\ \\ \text{TEMP, expr 1, expr 2}\end{Bmatrix}$$

$$\text{DBACKSPACE [\#f,]} \begin{Bmatrix}\text{BEG}\\ \\ \text{sectors [S]}\end{Bmatrix}$$

$$\text{DSKIP [\#f,]} \begin{Bmatrix}\text{END}\\ \\ \text{sectors [S]}\end{Bmatrix}$$

IF END THEN line no

$$\text{LIMITS} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \ \text{[\#f,] file-name, starting, ending, used}$$

LIMITS $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}$ [#f,] starting, ending, current

LIST DC $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$

LOAD DA $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$ (expr 1, return var) [line 1] [,line 2]

LOAD DC $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$ file-name [line 1] [,line 2]

MOVE $\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}\begin{Bmatrix}\text{FR}\\\\\text{RF}\end{Bmatrix}$

MOVE END $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$ = expr 2

SCRATCH $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$ file name [,file name]

SCRATCH DISK $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$ [LS=expr 1,] END = expr 2

VERIFY $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}\begin{bmatrix}\text{\#f,}\\\\\text{/xyy,}\end{bmatrix}$ (expr 1, expr 2)

COMMANDS

$$\text{LOAD DA}\left\{\begin{matrix}\text{F}\\\text{R}\\\text{T}\end{matrix}\right\}\left[\begin{matrix}\text{\#f,}\\ \\\text{/xyy,}\end{matrix}\right]\text{(expr 1, return var)}$$

$$\text{LOAD DC}\left\{\begin{matrix}\text{F}\\\text{R}\\\text{T}\end{matrix}\right\}\left[\begin{matrix}\text{\#f,}\\ \\\text{/xyy,}\end{matrix}\right]\text{file name}$$

$$\text{SAVE DA}\left\{\begin{matrix}\text{F}\\\text{R}\\\text{T}\end{matrix}\right\}\text{[\$]}\left[\begin{matrix}\text{\#f,}\\ \\\text{/xyy,}\end{matrix}\right]\text{[P] (expr 1, return var) [line 1] [,line 2]}$$

$$\text{SAVE DC}\left\{\begin{matrix}\text{F}\\\text{R}\\\text{T}\end{matrix}\right\}\text{[\$]}\left[\begin{matrix}\text{(space)}\\ \\\text{(old name)}\end{matrix}\right]\left[\begin{matrix}\text{\#f,}\\ \\\text{/xyy,}\end{matrix}\right]\text{[P] new name [line 1] [,line 2]}$$

GIO HOG MODE

To Hog Disk:

$$\text{\$GIO DISK HOG}\left\{\begin{matrix}\text{fiie number}\\\text{or}\\\text{disk device address}\end{matrix}\right\}\text{(4480, return var)}$$

To Release Disk:

$$\text{\$GIO DISK RELEASE}\left\{\begin{matrix}\text{file number}\\\text{or}\\\text{disk device address}\end{matrix}\right\}\text{(4480, return var)}$$

BASIC-2 SYNTAX

$$\text{COPY} \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \left[\begin{matrix}\text{\#f,}\\ \\ \text{/xyy,}\end{matrix}\right] \left[\text{(expr 1, expr 2)}\right] \text{ TO } \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \left[\begin{matrix}\text{\#f,}\\ \\ \text{/xyy,}\end{matrix}\right] \left[\text{(expr 3)}\right]$$

NOTES:

1) expr 1 and expr 2 optional
2) "TO" allows copy to a specified sector of separate disk unit

$$\text{DATALOAD BA} \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \left[\begin{matrix}\text{\#f,}\\ \\ \text{/xyy,}\end{matrix}\right] \text{(expr 1 [,return var]) alpha-array}$$

NOTE:

1) return variable optional

$$\text{DATALOAD DA} \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \left[\begin{matrix}\text{\#f,}\\ \\ \text{/xyy,}\end{matrix}\right] \text{(expr 1 [,return var]) arg-list}$$

NOTE:

1) return variable optional

DATALOAD DC [#f,] arg-list

$$\text{DATALOAD DC OPEN} \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \left\{\begin{matrix}\text{TEMP [,] expr 1, expr 2}\\ \text{\#f,}\\ \text{file name}\end{matrix}\right\}$$

NOTES:

1) comma after TEMP optional

2) file can be specified with a file number

DATASAVE BA $\left\{\begin{matrix}\text{F}\\\text{R}\\\text{T}\end{matrix}\right\}\left[\begin{matrix}\text{#f,}\\\\\text{/xyy,}\end{matrix}\right]$ (expr 1 [,return var]) $\left\{\begin{matrix}\text{alpha-variable}\\\\\text{literal string}\end{matrix}\right\}$

NOTES:

1) return variable optional

2) data can be saved specified as a literal string or alpha-variable

DATASAVE DA $\left\{\begin{matrix}\text{F}\\\text{R}\\\text{T}\end{matrix}\right\}$ [$] $\left[\begin{matrix}\text{#f,}\\\\\text{/xyy,}\end{matrix}\right]$ (expr 1, [return var]) $\left\{\begin{matrix}\text{END}\\\text{expression}\\\text{arg-list}\\\text{literal}\end{matrix}\right\}$

NOTE:

1) return variable optional

DATASAVE DC [$] [#f,] $\left\{\begin{matrix}\text{END}\\\text{arg-list}\\\text{expression}\\\text{literal}\end{matrix}\right\}$

DATASAVE DC CLOSE $\left[\begin{matrix}\text{#f,}\\\\\text{ALL}\end{matrix}\right]$

$$\text{DATASAVE DC OPEN} \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \text{ [\$] [\#f,] } \left\{\begin{matrix} \left\{\begin{matrix}\text{old name}\\ \text{space}\end{matrix}\right\}\text{, new name} \\ \text{TEMP}\left[,\right]\text{expr 1, expr 2}\end{matrix}\right\}$$

NOTE:

1) comma after TEMP optional

$$\text{DBACKSPACE [\#f,]} \left\{\begin{matrix}\text{BEG}\\ \text{sectors [S]}\end{matrix}\right\}$$

$$\text{DSKIP [\#f,]} \left\{\begin{matrix}\text{END}\\ \text{sectors [S]}\end{matrix}\right\}$$

$$\text{IF END THEN} \left\{\begin{matrix}\text{line no}\\ \text{statement}\end{matrix}\right\} \left[\text{:ELSE statement}\right]$$

NOTES:

1) if condition is true can also execute specified statement

2) "ELSE" syntax can specify statement to be executed if false

$$\text{LIMITS} \left\{\begin{matrix}\text{F}\\ \text{R}\\ \text{T}\end{matrix}\right\} \text{[\#f,] file name, starting, ending, used [,status]}$$

NOTE:

1) can also return status (type and condition) of file

LIMITS $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix}$ [#f,] starting, ending, current

LIST [S] [title] DC $\begin{Bmatrix}\text{F}\\\text{R}\\\text{R}\end{Bmatrix} \begin{bmatrix}\text{\#f}\\\text{/xyy}\end{bmatrix}$

NOTES:

1) can list in steps on CRT

2) prints title $\begin{matrix}\text{literal string}\\\text{or}\\\text{alpha variable}\end{matrix}$ in expanded print

LOAD DA $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix} \begin{bmatrix}\text{\#f,}\\\text{/xyy,}\end{bmatrix}$ (expr 1 [,return var]) [line 1] [,line 2] [BEG begin]

NOTES:

1) return variable optional

2) can begin execution at any line number specified

LOAD [DC] $\begin{Bmatrix}\text{F}\\\text{R}\\\text{T}\end{Bmatrix} \begin{bmatrix}\text{\#f,}\\\text{/xyy,}\end{bmatrix} \begin{Bmatrix}\text{file-name}\\\text{<\# files>alpha var}\end{Bmatrix}$ [line 1] [,line 2] [BEG begin]

NOTES:

1) can specify number of files to be loaded and begin execution at specified line no.

2) DC optional

$$\text{MOVE} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\#\text{f,}\\ \\ /\text{xyy,}\end{bmatrix} \text{[file-name] TO} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\#\text{f,}\\ \\ /\text{xyy,}\end{bmatrix} \begin{bmatrix}\text{(file name)}\\ \\ \text{(space)}\end{bmatrix}$$

NOTES:

1) can move entire platter or specified file between separate disk units

2) can reserve additional sectors for the new file

$$\text{MOVE END} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\#\text{f,}\\ \\ /\text{xyy,}\end{bmatrix} = \text{expr 2}$$

$$\text{SCRATCH} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\#\text{f,}\\ \\ /\text{xyy,}\end{bmatrix} \text{file name [,file name]}$$

$$\text{SCRATCH DISK} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\#\text{f,}\\ \\ /\text{xyy,}\end{bmatrix} \text{[LS=expr 1,] END = expr 2}$$

$$\text{VERIFY} \begin{Bmatrix}\text{F}\\ \text{R}\\ \text{T}\end{Bmatrix} \begin{bmatrix}\#\text{f,}\\ \\ /\text{xyy,}\end{bmatrix} \text{[(expr 1, expr 2)][numeric receiver variable]}$$

NOTES:

1) sector specification optional

2) can report address of next sector after that which did not verify

LOAD DA $\begin{Bmatrix} \text{F} \\ \text{R} \\ \text{T} \end{Bmatrix} \begin{bmatrix} \text{\#f,} \\ \text{/xyy,} \end{bmatrix}$ (expr 1, [return var])

> **NOTE**:
>
> 1) return variable optional

LOAD [DC] $\begin{Bmatrix} \text{F} \\ \text{R} \\ \text{T} \end{Bmatrix} \begin{bmatrix} \text{\#f,} \\ \text{/xyy,} \end{bmatrix}$ file name

> **NOTE**:
>
> 1) DC optional

LOAD RUN $\begin{Bmatrix} \text{F} \\ \text{R} \\ \text{T} \end{Bmatrix} \begin{bmatrix} \text{\#f} \\ \text{/xyy} \end{bmatrix}$ [file name]

SAVE DA $\begin{bmatrix} \text{<S >} \\ \text{<SR>} \end{bmatrix} \begin{Bmatrix} \text{F} \\ \text{R} \\ \text{T} \end{Bmatrix}$ [$] $\begin{bmatrix} \text{\#f,} \\ \text{/xyy,} \end{bmatrix} \begin{bmatrix} \text{!} \\ \text{P} \end{bmatrix}$ (expr 1 [,return var])[line 1] [,line 2]

> **NOTES**:
>
> 1) can save programs with spaces and remarks (REM) deleted
>
> 2) can save program scrambled
>
> 3) return variable optional

SAVE [DC] $\begin{bmatrix} \text{<S >} \\ \text{<SR>} \end{bmatrix} \begin{Bmatrix} \text{F} \\ \text{R} \\ \text{T} \end{Bmatrix}$ [$] $\begin{bmatrix} \text{\#f,} \\ \text{/xyy,} \end{bmatrix} \begin{bmatrix} \text{(old name)} \\ \text{(space)} \end{bmatrix} \begin{bmatrix} \text{!} \\ \text{P} \end{bmatrix}$ new name [line 1][,line 2]

NOTES:

1) can save programs with spaces and remarks (REM) deleted

2) can save program scrambled

3) DC optional

GIO HOG MODE

To Hog Disk:

$\$GIO\ DISK\ HOG \left\{ \begin{array}{c} \text{file number} \\ \text{or} \\ \text{disk device address} \end{array} \right\} (4480)$

To Release Disk:

$\$GIO\ DISK\ RELEASE \left\{ \begin{array}{c} \text{file number} \\ \text{or} \\ \text{disk device address} \end{array} \right\} (4480)$

Glossary of pertinent descriptions:

sectors	= number of sectors traversed expression
expr 1	= starting sector expression
expr 2	= ending sector expression
expr 3	= destination sector expression
arg-list	$= \left\{ \begin{array}{l} \text{alpha or numeric scalar} \\ \\ \text{alpha or numeric array} \end{array} \right\} \left[\left\{ \right. \right.$
space	= no. of sectors (additional) to reserve
starting	= numeric variable for starting sector address
ending	= numeric variable for ending sector address
current	= numeric variable for current sector address
used	= numeric variable for sectors contained
file name	= literal string or alpha variable up to 8 characters in length
# files	= the number of files used

APPENDIX C
A GLOSSARY OF DISK TERMINOLOGY

absolute sector address	— An address permanently assigned to a disk sector.
Absolute Sector Addressing Mode	— A mode of disk operation which enables the programmer to address individual sectors on disk. Also referred to as 'direct addressing' mode.
access	— See 'disk access' and 'file access'.
argument	— In a DATASAVE DC or DA statement, a discrete value, specified directly (as a numeric value or literal string in quotes) or indirectly (as the value of a variable or array element). Each argument occupies a single field in the record on disk, and is separated from neighboring fields by a Start-of-Value (SOV) byte. In a DATALOAD DC or DA statement, each receiving variable or array element which receives one value when the record is read from disk is regarded as a receiving argument. For the most part, multiple arguments in a statement must be separated by commas; however, when an array designator is used to specify an entire array, each element of the array is regarded as a separate argument.
argument list	— The list of all arguments in a DATASAVE DC/DA or DATALOAD DC/DA statement.
Automatic File Cataloging Mode	— A mode of disk operation in which the names and locations of files on disk are maintained automatically by the system in a Catalog Index.
binary address	— A sector address expressed as a two-byte binary number.
binary search	— A dichotomizing search in which the number of records in the file is divided into two equal parts at each step in the search.
blocked records	— Two or more short records stored in one sector. Since the minimum length of any record is, from the system's point of view, one sector, the blocking of multiple records in a single sector must be a function of user's software.
Catalog Area	— The area on a disk platter reserved for the storage of cataloged files.
Catalog Index	— An index containing names and pointers for each cataloged file in the Catalog Area.
command	— A BASIC-2 instruction which provides the operator with control of a system function directly from the keyboard. Commands are entered and executed immediately by the operator; they cannot be stored in memory as part of a program.

control byte	—	Any of several special bytes created automatically by the system to help it keep track of data stored on the disk, and which are completely transparent to the user's software. See also 'start-of-value control byte' and 'sector control byte'.
cyclic redundancy check	—	A special checksum test automatically performed by the disk unit on all data read from the disk. Abbreviated CRC.
data file	—	A collection of related data records treated as a logical unit. For example, an inventory file contains a number of inventory records, each of which in turn consists of specified items of information about a particular item in the inventory. In catalog mode, a data file can be opened or reopened by name.
data record	—	See 'logical data record'.
default address	—	The device address for a System 2200 peripheral which is used automatically by the system when no other address is specified for the disk unit, the system default address is 310. The disk default address is always stored opposite the default file number (#0) in the Device Table, and may be changed temporarily with a SELECT DISK statement. However, the system default address (310) is automatically returned to the default slot upon Master Initialization. See also 'device address'.
default file number	—	The file number in the Device Table automatically used by the system when a disk statement or command is executed which does not specify a file number. The default file number is always #0, and cannot be changed. The default disk address is always stored in the slot opposite the default file number. See also 'default address', 'Device Table', and 'file number'.
device address	—	A three-digit hexadecimal code used by the CPU to identify each peripheral device. The device address is set in the controller board for each peripheral either at the factory or by a Wang Service Representative, and should be clearly printed on the top surface of the controller board. See also 'default address', 'device type', and 'unit device address'.
Device Table	—	A special section of memory used to store disk device addresses and sector address parameters for currently opened files on disk. It consists of 16 rows, or "slots", identified by file numbers #0 - #15. A device address and a set of sector address parameters for an open file can be stored in each slot. The slots opposite file numbers #1 - #15 are also used for other I/O devices in addition to the disk (such as paper tape readers, and card readers). The default slot (opposite #0) is used only for the disk, however. Default addresses for other I/O devices are stored in another section of memory. See also 'default address', 'default file number', and 'file number'.
device type	—	The first digit of the three-digit device address. For the disk unit, the device type can be either '3' or 'B' or 'D' (e.g., 3XX, or BXX). When used with the Model 2260 or 2270 series in conjunction with the 'T' parameter, the device type determines which disk platter in a multi-platter disk unit is to be accessed. In this case, a device type of '3' identifies the 'F' disk platter, while a device type of 'B' identifies the 'R' disk platter. For the Model 2270-1 Single Removable Diskette Drive, and for the third platter of the Models 2270-3 and 2270A-3, a device type of 'B' is illegal. See also device address' and 'unit device address'.

disk access	—	Any disk read or write operation. See also 'file access'.
disk drive	—	1. Broadly, a disk unit containing one or more disk platters.
	—	2. More specifically, the assembly (consisting of drive motor, spindle, and access arm(s)) which drives the disk platter(s) and is activated by a single disk command. See also 'disk platter'.
disk latency period	—	The period of time which elapses from the time the read/write head positions itself to a track until the desired sector in that track rotates to the read/write head's position. Disk latency time is determined by the rotation speed of the disk unit. Latency time may be important for random access operations; it is generally negligible in sequential access operations. See also 'track access time'.
disk platter	—	The flat, circular plastic or metal plate which is coated on its recording surface with a magnetic substance such as iron oxide, and which serves as the storage medium in a disk unit.
ending sector address	—	The address of the last sector in a file or multi-sector logical record. See 'starting sector address' and 'absolute sector address'.
end-of-file trailer	—	A special record, one sector in length, which marks the end of currently stored data in a data file. The end-of-file record is created with a DATASAVE DC END or DATASAVE DA END statement. Creation of an end-of-file trailer record in a cataloged file automatically causes the 'used' column in the Catalog Index to be updated, and enables the programmer to check for the end-of-file with an IF END THEN statement, or to skip to the end-of-file of a cataloged file with a DSKIP END statement.
expression	—	A numeric value (e.g., '1234'), operation (e.g., 'A*B 2'), variable (e.g., 'N') or array element (e.g., 'N(3)').
field	—	1. An individual item of data within a logical data record on the disk. Each argument in the DATASAVE DC or DATASAVE DA argument list is recorded as a single field (marked off by SOV control bytes) in the logical record created by the statement.
		2. A specified section of a record reserved for a particular type of information. For example, a 'key field' consists of a number of bytes located at a specific place in a record which always holds the key value for the record.
file	—	A collection of related records treated as a logical unit. Files may be of two types, program files and data files. In catalog mode, files can be created and accessed by name. See 'data file' and 'program file'.
file access	—	1. Any disk operation in which information (programs or data) is read from or written in a file on disk.
		2. Any disk operation which results in positioning the read/write head to a location preparatory to reading or writing information in a file. See also 'disk access'.

file number	—	One of the 16 numbers #0 - #15 associated with slots in the Device Table, and used to identify currently opened files on disk. File numbers are also used to identify non-disk files. A file number is always preceded by a "#" symbol. See 'default file number' and 'Device Table'.
hashing technique	—	A technique for storing and accessing information on disk in which a specialized algorithm, called a "hash function", is used to convert a record's key value into an absolute sector address, which is then used as the location at which the record is stored. This technique is used by the system in catalog mode to store file names in the Catalog Index.
header record	—	A record containing special control information and preceding all other records in a file. Every program file saved on disk begins with a one-sector header record. In cataloged programs, the header contains the program name, along with catalog system control information. Data files on disk have no header record, but cataloged data files do have a system control record at the end of the file which serves the same purpose as a header. See 'trailer record' and 'system control record'.
Hog Mode	—	A mode of disk multiplexer operation in which one station obtains exclusive access to the disk, while all other stations are locked out.
key field	—	A field in a record on the disk consisting of one or more bytes, and containing the key value for that record. See 'field' and 'key value'.
key value	—	A numeric or alphanumeric value in a record used to identify the record for purposes of access and control. See 'key field', 'sort', and 'hashing technique'.
logical data	—	A data record on the disk created by a record DATASAVE DC or DATASAVE DA statement which occupies one or more sectors, and contains all of the data from the DATASAVE DC or DATASAVE DA argument list. See also 'record' and 'data file'.
Logical Platter Address	—	A three digit hexadecimal code used by the CPU to identify each recording surface on the Model 2280 Disk Drive.
logical record	—	See 'logical data record'.
longitudinal redundancy check	—	A checksum test performed by the system on each sector of data read from the disk. Abbreviated LRC.
multiplexing	—	A process of allocating disk time to a number of systems by sequentially interleaving disk operations from the various inquiring systems.
multi-volume	—	A file occupying two or more disk platters. Each separate platter is considered a different "volume" of the file. Each volume must be carefully identified with a file name and a volume number.
parameter	—	An element in a BASIC statement or command which follows the BASIC verb, and whose function and meaning are defined for the purposes of the statement. Parameters may be of two types, constant (or fixed) and variable. The value of a fixed parameter is predefined and cannot be altered by the user. The value of a variable parameter is specified by the user, although there are nor-

		mally certain limitations imposed upon the range of values which may be assigned to a particular parameter. A fixed parameter is always indicated in the general form of a statement or command as an uppercase letter (e.g., 'P', 'DC', 'S', etc.), while a variable parameter is indicated with a lowercase letter (e.g., 'xxx', 'n') or described with a lowercase literal string (e.g., 'name', 'sector address', etc.).
pointer	—	An absolute sector address or displacement which "points" to the location of a record on the disk.
program file	—	A file on disk consisting of a single BASIC program or program segment, and optionally also containing extra sectors reserved for possible future expansion of the program. A program file always begins with a header record and ends with a trailer record. In catalog mode, a program file can be saved and loaded by name.
program record	—	A sector in a program file between the header record and the trailer record which contains program text. See 'header record' and 'trailer record'.
protect parameter	—	A special parameter ('P') or (!) used to protect programs saved on disk.
protected program	—	A program on disk which can be loaded and run, but cannot be listed or re-saved.
read-after-write verification	—	An optional verification check which can be performed on each sector of data as it is written on the disk. The read-after-write check is specified by including the dollar sign ('$') parameter in a disk statement or command. However, a read-after-write check significantly increases the execution time of the disk operation.
read/write head	—	An electromagnetic recording head which reads and writes information on the recording surface of a disk platter.
record	—	A collection of related items of data treated as a logical unit. See 'logical data record' and 'data file'.
sector	—	The basic unit of storage on a disk platter, consisting of a data field with a fixed length of 256 bytes, an absolute sector address, and certain control information. Each sector is regarded as a discrete unit, and is directly accessible by the system.
sector control bytes	—	Special control bytes containing system bytes control information which are written automatically by the system into each sector of a logical data record and each program record stored on disk. Each sector in a logical data record contains three sector control bytes; each one-sector program record in a program file contains two sector control bytes. The sector control bytes are transparent to the user's software.
sort	—	1. To arrange data sequentially in ascending or descending order. 2. To sequentially order logical data records in a file based upon the key values of the records. 3. The act of performing a sorting operation.

starting sector	— The address of the first sector in a address file or multi-sector logical record. See also 'ending sector address'.
start-of-value control byte	— A control byte created automatically by the system independent of user software, and prefixed to each field in a logical record when the record is written with a DATASAVE DC or DATASAVE DA statement. This control byte separates fields within a record and marks the beginning of each new field. The start-of-value bytes are not automatically written when a DATASAVE BA statement is executed. Abbreviated SOV.
statement	— Broadly, a generic term for all BASIC-2 programmable instructions. Every line in a BASIC-2 program consists of one or more statements, each of which directs the system to perform a specific operation or sequence of operations. Although statements are, by definition, programmable instructions, most statements also can be executed in Immediate Mode simply by entering them without a preceding line number.
system control record	— A special record one sector in length which always occupies the last sector of a cataloged data file, and contains control information and pointers for the file. A system control record is automatically created and updated by the system for each data file maintained in catalog mode; it is completely transparent to the user's software.
temporary files	— Files established outside the Catalog Area on a disk, generally for the storage of transient data. Temporary files cannot be named, and no entry is listed for them in the Catalog Index. They can, however, be accessed with catalog procedures.
track	— Any of the concentric circular electromagnetic paths into which the recording surface of a disk platter is divided. Each track, in turn, is subdivided into a number of sectors. The number of tracks on a platter differs according to the disk model and configuration. See 'sector' and 'disk platter'.
track access time	— The time required for the access assembly to move the read/write head from its current position to the track containing the desired sector. For random access operations, the track access time may become significant if the sectors to be accessed are scattered on widely separated tracks. For most sequential access operations, however, the track access time is negligible. See also 'disk latency time'.
trailer record	— 1. In program files, the sector immediately following the last program record. The trailer record contains control information, written automatically by the system, along with the last few lines of program text. 2. In data files, a special record created by specifying the 'END' parameter in a DATASAVE DC or DATASAVE DA statement, to mark the limit of valid data in the file. Also referred to as an "end-of-file" trailer record. See 'end-of-file trailer record'.
unit device address	— The last two digits of the three-digit device address (e.g., X10, X20, X50, etc.), which identify individual disk units when more than one is attached to the same system. See 'device address', and 'device type'.
work files	— See 'temporary files'.

APPENDIX D
BIBLIOGRAPHY

The techniques involved in creating, maintaining, and accessing disk-based data files are the subjects of an extensive number of textbooks and articles. The authors included in this bibliography approach the programming problems associated with disk storage from a variety of different perspectives, and with varying degrees of sophistication. In general, however, the bibliography has been heavily weighted toward the relative novice, although in all cases some background in programming is required.

It is suggested that the programmer with little or no experience in disk operations begin with a text which provides a general survey of the standard types of disk file structures and access techniques. (The titles identified with asterisks provide such a survey at an introductory or intermediate level.) The number of disk storage and access techniques which have been developed over the last 10 or 15 years is considerable, even if one restricts oneself only to the "standard" techniques, and each has particular strengths and weaknesses which make it suitable for some applications and most unsuitable for others. Armed with an overview of the available systems and techniques, the programmer will be in a position to determine which of them most appropriately suit his own application. He then can proceed to a textbook or article which treats the chosen technique(s) in greater depth.

1. Bosco, R.L., Data Bases, Computers, and the Social Sciences (Wiley-Interscience, New York, 1970).

2. Brooks, F.P., and K.E. Iverson, Automatic Data Processing (John Wiley and Sons, New York, 1963).

3. Clemenson, W.D., "File Organization and Search Techniques," Annual Review of Information Science and Technology, Volume 1, Ed. C. Cuadra (John Wiley and Sons, New York, 1966).

4. Daley, R.C., and P.G. Newmann, "A General Purpose File System for Secondary Storage," Proceedings of the AFIPS 1965 Fall Joint Computer Conference, Volume 27, Part 1 (Spartan Books, New York).

5. Dodd, G.G., "Elements of Data Management Systems," Computer Surveys, Volume 1, No. 2, June 1966.

*6. Forsythe, A.I., and T.A. Keenan, E.I. Organick, and W. Stenberg, Computer Science: A First Course (John Wiley and Sons, New York, 1969).

7. Gear, C.W., Computer Organization and Programming (McGraw Hill, New York, 1969).

* Titles marked with an asterisk are intermediate-level texts recommended for programmers with limited background in disk operations.

8. Gruenberger, F. (Ed.), Critical Factors in Data Management (Prentice-Hall, Englewood Cliffs, N.J., 1969).

9. Hsiao, D. and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Volume 13, Number 2 (February, 1970).

10. Hull, T.E. and D.F. Day, Computers and Problem Solving (Addison-Wesley (Canada) Ltd., Don Mills, Ontario, 1970).

11. Iverson, K.E., A Programming Language (John Wiley and Sons, New York, 1962).

12. Johnson, L.R., "Indirect Chaining Method for Addressing on Secondary Keys," Communications of the ACM, Volume 4, Number 4 (May 1961).

13. Korfhage, R.R., Logic and Algorithms (John Wiley and Sons, New York, 1966).

14. Knuth, D.E., The Art of Computer Programming, Volumes I and III (Addison-Wesley, Reading, Mass., 1968).

*15. Lefkowitz, D., File Structures for On-Line Systems (Spartan Books, New York, 1969).

16. Lowe, T.C., "The Influence of Data-Base Characteristics and Usage on Direct Access File Organization," Journal of the ACM, Volume 15, Number 4 (October, 1968).

17. Martin, J., Design of Real-Time Computer Systems (Prentice-Hall, Englewood Cliffs, N.J., 1967).

18. Mauer, W.D., "An Improved Hash Code for Scatter Storage," Communications of the ACM, Volume 11, Number 1 (January, 1968).

19. McIlroy, M.D., "A Variant Method of File Searching," Communications of the ACM, Volume 6, Number 1 (January, 1963).

20. Meadow, C.T., The Analysis of Information Systems (John Wiley and Sons, New York, 1967).

21. Morris, R., "Scatter Storage Techniques," Communications of the ACM, Volume 11, Number 1, (January, 1968).

22. Peterson, W.W., "Addressing for Random-Access Storage," IBM Journal of Research and Development, Volume 1 (1957).

23. Rosove, P.E., Developing Computer-Based Information Systems (John Wiley and Sons, New York, 1967).

24. Williams, W.F., Principles of Automatic Information Retrieval (The Business Press, Elmhurst, Illinois, 1968).

*25. Yourdon, E., Design of On-Line Computer Systems (Prentice-Hall, Englewood Cliffs, N.J., 1972).

* Titles marked with an asterisk are intermediate-level texts recommended for programmers with limited background in disk operations.

APPENDIX E
DISK FILE BACK-UP

Probably the most common form of the file security is file backup. File backup is simply maintaining a backup copy of important files. Backing up important files is an area that should be given high priority.

Disk storage devices are basically very reliable. However, like any other storage media, disk platters are subject to accidental damage or destruction. Losing power during an update, dropping a disk cartridge, exposing it to a magnetic device are just a few of the things that could cause the destruction of data.

Most computer users cannot afford the cost and inconvenience associated with the reconstruction of a destroyed disk file. Some companies have been severely crippled when a critical file was accidentally destroyed because they did not adhere to rigid backup procedures for all essential programs and data files.

Many small computer users think that the cost and time associated with maintaining backup files is high. The only cost associated with file backup is the price of an extra storage device such as a disk platter or diskette. The time involved backing up a file is minimal when the alternatives are considered.

Frequency

There is no absolute rule governing backup frequency. It normally depends upon several factors. One important factor is the amount of activity or the number of transactions processed against a master file. With high activity, a user may wish to back up a data file daily. With fewer transactions, a frequency of once or twice a week may be sufficient.

Another important factor is time. For example, to back up a full 2260 disk platter can take up to 30 minutes. Therefore, this time should be weighed against the time it would take to reconstruct the data if the file were destroyed. While doing so, the user should also consider the overall effect the time spent reconstructing a critical file would have on his business.

Each user, therefore, should carefully evalute the factors relating to his own business and data processing requirements. As a general rule, it is recommended that important files be backed up with a frequency that matches the processing activity of that file. In other words, if a file is updated daily, it should be backed up daily; if a file is updated weekly, it should be backed up weekly.

It is also a good practice to create an extra copy of a backup platter and/or keep more than one generation of these platters. Very often, the backup platter can be ruined by the same problem that destroyed the original platter. Having this extra backup platter provides an additional measure of protection against time-consuming and costly data reconstruction.

Some companies also periodically store backup files at an off-premise location. By doing so, they protect themselves against the danger of fire, explosions, or some other disaster.

Procedure

The procedure for backing up files varies from one application to another. With Wang-provided application software, it is normally a matter of mounting a backup diskette or disk platter, loading the proper module, and following the instruction prompts provided.

A user may back up a disk platter in one of the three following ways:

1. COPY statement
2. MOVE statement
3. COPY/VERIFY Utility

COPY Statement

The COPY statement copies the entire contents of a disk platter, or a specified portion of its contents, to another disk platter in the same or different disk unit. The COPY statement is the only "Absolute Sector Addressing" mode, BASIC-2 statement that should be used in backing up a file.

Example:

```
10 COPY F (0, 2000) TO R
20 VERIFY R (0, 2000)
```

Statement 10 copies sectors zero through 2000 from the fixed (F) platter to the same sectors on the removable (R) platter. Statement 20 verifies through longitudinal and cyclical redundancy checking (LRC + CRC) that the data recorded on the backup disk cartridge is valid.

Starting and ending sector addresses of the information to be copied should always be included in the COPY statement. If the entire contents of a disk platter are copied, the beginning sector address should be zero and the ending address should be the last sector on the platter.

If an error is encountered following a COPY operation, the process should be repeated. Repeated failure could indicate a faulty disk platter. If the error persists with another platter, a Wang Service Representative should be called.

Additional information concerning the use of the COPY statement can be found in Chapter 7 of this manual.

MOVE Statement

The MOVE statement, used only with cataloged files, provides another means of backing up disk files. In addition to copying the catalog index and data files, it also provides one additional function. The MOVE statement eliminates scratched files from the catalog and compresses still-active files into the available space.

Since it only copies active files, the MOVE statement results in a faster copy than the COPY statement. However, caution should be exercised when using MOVE that only cataloged files, are on the disk platter. Any other files will be lost unless a COPY statement is used.

Example:

```
10 MOVE F TO R
20 VERIFY R
```

Statement 10 copies all catalog information from the "F" disk platter to the "R" disk platter. Statement 20 checks the "R" disk platter to ensure that all information has been copied correctly.

Additional information concerning the use of the MOVE statement can be found in Chapters 2 and 5 of this manual.

When using either the COPY or MOVE statements on the Model 2260 and 2270 Series, it is very critical that the "F" and "R" parameters are positioned correctly. Reversing these two characters will destroy the original files. To avoid this occurrence, a small utility can be written, incorporating the MOVE and COPY statements, which provides the necessary prompts on the CRT and other safeguards to prevent the accidental destruction of a disk platter that is to be copied.

COPY/VERIFY Utility

There is one other way to back up important files. This final method utilizes the ISS utility COPY/VERIFY and can be used by those customers who have purchases Wang's Integrated Support System (ISS) software packages.

The COPY/VERIFY utility offers more flexibility than the COPY and MOVE statements and offers the following features:

1. Copied files may be renamed and may replace existing files on the output disk.
2. Selected files or all files may be copied without altering files on the output platter.
3. Copying is allowed between any two platter/disk addresses.
4. Copying is accomplished by read/write operations rather than COPY or MOVE statements.
5. The verify operation actually compares the data read from the input file to the data written on the output file to insure that it has been copied correctly.
6. Additional sectors may be added to the copied file.

The operating instructions for the COPY/VERIFY utility are outlined in the *Integrated Support System User Manual.*

With all diskette devices, accidental destruction of data can be avoided by the proper use of the Write Protect feature. A small notch along the edge of the diskette's plastic jacket controls the Write Protect mechanism. When this notch is uncovered, the diskette is (write) protected. No information can be recorded on it, nor can it be formatted.

In conclusion, file backup is extremely important to all Data Processing installations. Unless adequate precautions are taken now, serious consequences may result later. We hope this discussion will help avoid any serious and costly problems resulting from inadequate backup.

APPENDIX F
MODEL 2280 DISK MULTIPLEXER

With the Model 2280 Disk Multiplexer, any combination of one, two, or three CPUs (2200MVP, LVP, or VP) can share a Model 2280 Disk Drive or pair of 2280 disk drives (i.e., 2280 and 2280N).

When more than one CPU is configured to access a common disk data base, a multiplexed disk environment exists. Multiplexing adds an important dimension to disk ownership. A single disk unit can be apportioned among several offices or departments. Each user can have access to a common disk data base while retaining an individual CPU in a convenient location. The disk operations from multiple-inquiring systems are interleaved, and disk time is allocated among the inquiring CPUs in a manner that provides all CPUs with virtually concurrent access to the disk.

DISK MULTIPLEXER CONFIGURATION

The Model 2280 Disk Multiplexer is a "star-type" multiplexer consisting of the multiplexer board, individual connector cables to each of up to three CPUs, and one controller board for each CPU in the configuration. The Model 2280 multiplexer installs directly into a 2280 Multiplexable Disk Processing Unit (MDPU) and contains the polling circuitry, the interface to the DPU, and three ports for cable connection to the 2200 CPUs. (Some older DPUs require updates to the motherboard before multiplexing can be supported.) Each participating CPU must have a Model 22C80 controller installed in its I/O bus. (The 22C03 Disk Controller, 22C11 Dual Controller, or 22C32 Triple Controller *cannot* be used to interface the 2280 multiplexer.) A standard 12-foot (3.7 meters) connector cable is shipped with each Model 22C80 controller board for connection to the multiplexer board.

In addition to the standard 12-foot (3.7 meters) connector cable, extension cables are available in lengths of 25, 50, 100, 250, 500, 750, and 1000 feet (7.6, 15.2, 30.3, 75.8, 151.5, 227.3, and 303 meters). The following list contains the available extension cable lengths and their appropriate part numbers.

Cable Length		**Part Number**
25 ft	(7.6 m)	120-2280-01
50 ft	(15.2 m)	120-2280-02
100 ft	(30.3 m)	120-2280-03
250 ft	(75.8 m)	120-2280-04
500 ft	(151.5 m)	120-2280-05
750 ft	(227.3 m)	120-2280-06
1000 ft	(303.0 m)	120-2280-07

These cables can only serve as extensions for the standard connector cables; an extension cable cannot be used by itself to connect a DPU to a CPU. Because the extension cable is combined with the standard cable in this manner, the total length of the cable between two units is always equal to the extension cable length plus 12 feet. In special cases, it is possible to connect two or more extension cables to create an extension longer than 1000 feet. In every case, however, the maximum permissible distance between the DPU and a CPU is 1012 feet.

INSTALLING THE MODEL 2280 DISK MULTIPLEXER

Unpacking and Inspection

Because the Model 2280 Disk Multiplexer unit is a sensitive device, it is packed using special techniques to protect it from damage in shipping. It should be unpacked and inspected only by a qualified Wang Service Representative. Failure to follow this procedure voids the Wang equipment warranty.

Installation/Power-on Procedure

CAUTION

Wang Laboratories, Inc., does not guarantee any equipment modified by the user. Damage to equipment incurred as a result of such modificaton is the financial responsibility of the user. It is recommended that only Wang Service Representatives modify Wang equipment. Contact your Wang Field Service Office to perform all installation services.

To install the disk multiplexer and to power on the system, use the following procedure.

1. Power down all system peripherals, including disk units, all terminals, and printers. Switch off the main power switch of the DPU, followed by the main power switch of the CPU.
2. Install the Model 2280 multiplexer board into the DPU. Connect the cable between the DPU and the multiplexer board. (Refer to Figure F-1.)
3. Set the proper disk address on each Model 22C80 controller (usually 10, 20, or 30). No disk controller boards in the same CPU should have the same address. Install a 22C80 controller board in each participating CPU.
4. Plug one end of the disk I/O connector cable into the 22C80 controller board. Plug the other end of this connector cable into an available port on the 2280 multiplexer board.
5. Repeat Step 4 for all attached CPUs.
6. Be sure that all attached systems are properly set up and ready for operation and all power cords are plugged into grounded (3-hole) wall sockets.
7. Follow the normal power-on procedure discussed in the *Model 2280/2280N Disk Drive User Manual* (700-5216A).

When routing the multiplexer connector cables between participating systems, take care to avoid exposing a cable to intense electric or magnetic fields, or sources of electronic noise, since they may interfere with data transmission over the cable.

If you have difficulty in maintaining valid data transmission between the disk and one or more systems, the problem may lie in the connector plugs. A coating sometimes forms on the pins of a plug during extended periods of disuse. To remove this coating, simply insert and remove the plug in a jack several times, or cut a piece from an ink-type eraser small enough to fit between the pins, and use it to clean the surfaces of the pins.

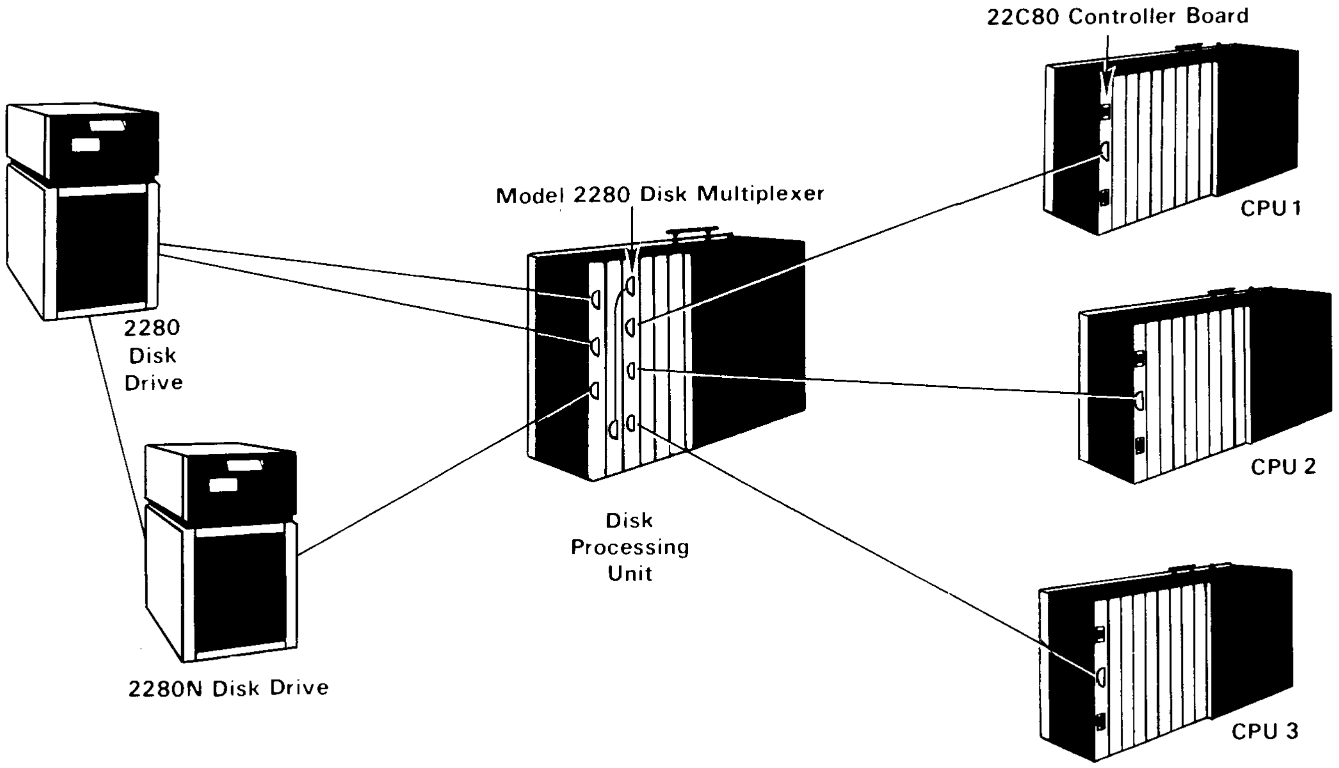


Figure F-1. A Multiplexed 2280 Disk Drive

MULTIPLEXER OPERATION

The disk multiplexer controls all communication between participating CPUs and the disk unit. All CPUs connected to a 2280MUX multiplexing DPU have equal access priority. Polling is done by sequentially scanning each port. The multiplexer automatically "polls" each CPU, beginning with CPU 1, until it finds a CPU attempting to access the disk. At that point, the multiplexer permits the inquiring CPU to execute one disk statement or command. Following execution of the statement or command, the multiplexer resumes its polling until it encounters another CPU trying to access the disk. The multiplexer does not monitor the amount of time required to execute each statement, nor does it limit the number of sectors transferred by a statement. A single statement may read or write only one sector, but it is equally possible to carry out multisector transfers with one statement. (A MOVE or COPY statement, for example, might transfer an entire disk platter to a second platter.) It is recommended, however, that major file maintenance operations be executed only by a CPU in Hog mode (refer to Section F.5). In any case, the CPU that is executing the statement retains use of the disk until statement execution is completed. Control is then transferred to the next inquiring CPU. The Model 2280MUX provides no external indication of which system has access to the disk.

In normal operation, the multiplexer imposes no special demands or conditions upon the programmer. The disk is simply addressed as usual with the appropriate disk statements and commands. (All disk addressing is identical to existing 2280 Disk Drives.) If no other CPUs are accessing the disk, the total execution time of a multistatement disk operation is not noticeably affected by the multiplexer. If more than one multistatement disk operation is being carried on at once, however, the time required for each operation is roughly equal to the total time required to execute all operations, since one statement from each CPU is executed on each pass by the multiplexer.

Multiplexing is transparent to user software except for the following.

1. Response time is degraded according to the disk load.

2. Single-user software must $OPEN the disk during critical periods when exclusive disk use is required.

HOG MODE OPERATION

Although in general all CPUs attached to the multiplexer gain access to the disk on a statement-by-statement basis, there are cases in which it is desirable to give one CPU a period of exclusive and uninterrupted access to the disk. During certain critical file maintenance or update procedures, for example, it is important that other CPUs be prevented from accidentally interfering in the routine, since they might unknowingly overwrite valuable data or pointers, or otherwise confuse the situation. Because operators on remote terminals have no way of knowing that critical maintenance procedures are being carried out at any given time, it is necessary to prevent them from accidentally interrupting a routine by locking them out. A CPU that monopolizes the disk in this way is said to be "hogging" the disk. *Every* disk platter in the disk unit is hogged when the disk unit is hogged. Whenever a CPU is granted access to a disk platter, it automatically gains control of all platters associated with that disk drive. A terminal that has the disk in Hog mode can execute any number of disk statements or commands while maintaining exclusive control of the disk, preventing other terminals from executing any operation on the hogged disk drive.

The following statements are recommended for Hog mode on the 2200MVP, LVP, and VP (Release 1.8 or later).

- $OPEN disk device address (to hog the disk)

- $CLOSE disk device address (to release the disk)

The following program illustrates the $OPEN and $CLOSE statements used to hog the disk. The user should refer to the *BASIC-2 Language Reference Manual* for an indepth discussion of these two statements.

```
110 REM OPEN FILE IN NON-HOG MODE
120 SELECT #1/D20
130 DATA LOAD DC OPEN T#1, "DATAFILE"
 .
 .    (processing)
 .
270 DBACKSPACE #1, BEG
280 DSKIP #1, N S : REM SKIP N SECTORS
290 REM UPDATE RECORD IN HOG MODE
300 $OPEN #1 :REM ENTER HOG MODE
310 DATA LOAD DC #1, A,B,C :REM READ RECORD
```

```
320 DBACKSPACE #1, 1 S
330 DATA SAVE DC #1, A, B+K, C:REM UPDATE
340 $CLOSE #1 :REM LEAVE HOG MODE
```

This program illustrates a typical update routine in which Hog mode is activated temporarily during the actual updating (from the time the record is read until its updated version is written). The file is opened with the disk drive in Non-hog mode (Line 130). Lines 270 and 280 locate the desired record also while in Non-hog mode. Hog mode is entered upon execution of Line 300. The multiplexer ceases its polling of the CPUs upon entering Hog mode. This terminal maintains exclusive access to the entire disk drive until executing Line 340, when Hog mode is left. (The hogging terminal also loses control of the disk drive if RESET is pressed on the terminal keyboard.)

The following points should be noted in regard to the operation of Hog and Non-hog mode.

1. When a multiplexed disk drive is hogged, the entire disk unit (all platters) is hogged.

2. Only the terminal which activates Hog mode can deactivate it.

3. If a terminal attempts to execute a disk statement while another terminal is hogging the disk drive, the terminal simply waits, with the processing light on, until Hog mode is released.

4. Hog mode is deactivated if RESET is pressed at the hogging terminal.

INDEX

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

700-4081G

TITLE OF MANUAL **WANG BASIC-2 DISK REFERENCE MANUAL**

COMMENTS:

Fold

Fold

(Please tape, Postal regulations prohibit the use of staples.)

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 16 LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Attention: Technical Writing Department

Fold

Cut along dotted line.

Printed in U.S.A.
13-1019

International Representatives

Argentina
Bahamas
Bahrain
Bolivia
Botswana
Brazil
Canary Islands
Chile
Colombia
Costa Rica
Cyprus
Denmark
Dominican Republic
Ecuador
Egypt
El Salvador
Finland
Ghana
Greece
Guam
Guatemala
Haiti
Honduras
Iceland
India
Indonesia
Ireland
Israel
Italy
Ivory Coast
Japan
Jordan
Kenya
Korea
Kuwait
Lebanon
Liberia
Malaysia
Malta
Mexico
Morocco
New Guinea
Nicaragua
Nigeria
Norway
Paraguay
Peru
Philippines
Portugal
Qatar
Saudi Arabia
Scotland
Senegal
South Africa
Spain
Sri Lanka
Sudan
Tasmania
Thailand
Turkey
United Arab Emirates
Uruguay
Venezuela
Zimbabwe

United States

Alabama
Birmingham
Mobile
Alaska
Anchorage
Juneau
Arizona
Phoenix
Tucson
California
Anaheim
Burlingame
Culver City
Emeryville
Fountain Valley
Fresno
Los Angeles
Sacramento
San Diego
San Francisco
Santa Clara
Ventura
Colorado
Englewood
Connecticut
New Haven
Stamford
Wethersfield
District of Columbia
Washington

Florida
Coral Gables
Hialeah
Hollywood
Jacksonville
Miami
Orlando
Sarasota
Tampa
Georgia
Atlanta
Savannah
Hawaii
Honolulu
Maui
Idaho
Boise
Illinois
Arlington Heights
Chicago
Morton
Oakbrook
Park Ridge
Rock Island
Rosemont
Springfield
Indiana
Fort Wayne
Indianapolis
South Bend

Iowa
Ankeny
Kansas
Overland Park
Wichita
Kentucky
Louisville
Louisiana
Baton Rouge
Metaine
Maine
Portland
Maryland
Baltimore
Bethesda
Gaithersburg
Rockville
Massachusetts
Boston
Burlington
Chelmsford
Lawrence
Littleton
Lowell
Methuen
Tewksbury
Worcester
Michigan
Grand Rapids
Kalamazoo
Lansing

Southfield
Minnesota
Eden Prairie
Minneapolis
Mississippi
Jackson
Missouri
Creve Coeur
St. Louis
Nebraska
Omaha
Nevada
Las Vegas
New Hampshire
Manchester
New Jersey
Bloomfield
Clifton
Edison
Mountainside
Toms River
New Mexico
Albuquerque
Santa Fe
New York
Albany
Jericho
Lake Success
New York City
Rochester

Syossett
Syracuse
Tonawanda
North Carolina
Charlotte
Greensboro
Raleigh
Ohio
Akron
Cincinnati
Cleveland
Independence
Toledo
Worthington
Oklahoma
Oklahoma City
Tulsa
Oregon
Eugene
Portland
Salem
Pennsylvania
Allentown
Erie
Harrisburg
Philadelphia
Pittsburgh
State College
Wayne
Rhode Island
Providence

South Carolina
Charleston
Columbia
Tennessee
Chattanooga
Knoxville
Memphis
Nashville
Texas
Austin
Dallas
El Paso
Houston
San Antonio
Utah
Salt Lake City
Virginia
Newport News
Norfolk
Richmond
Rosslyn
Springfield
Washington
Richland
Seattle
Spokane
Wisconsin
Appleton
Brookfield
Green Bay
Madison
Wauwatosa

International Offices

Australia
Wang Computer Pty., Ltd.
Adelaide, S.A.
Brisbane, Qld.
Canberra, A.C.T.
Perth, W.A.
South Melbourne, Vic 3
Sydney, NSW
Austria
Wang Gesellschaft, m.b.H.
Vienna
Belgium
Wang Europe, S.A.
Brussels
Erpe-Mere
Canada
Wang Canada Ltd.
Burlington, Ontario
Burnaby, B.C.
Calgary, Alberta
Don Mills, Ontario
Edmonton, Alberta
Halifax, Nova Scotia
Hamilton, Ontario
Montreal, Quebec
Ottawa, Ontario
Quebec City, Quebec
Toronto, Ontario
Victoria, B.C.
Winnipeg, Manitoba
China
Wang Industrial Co., Ltd.
Taipei
Wang Laboratories, Ltd.
Taipei
France
Wang France S.A.R.L.
Paris
Bordeaux
Lille
Lyon
Marseilles
Nantes
Nice
Rouen
Strasbourg
Great Britain
Wang (U.K.) Ltd.
Richmond
Birmingham
London
Manchester
Hong Kong
Wang Pacific Ltd.
Hong Kong

Japan
Wang Computer Ltd.
Tokyo
Netherlands
Wang Nederland B.V.
IJsselstein
Groningen
New Zealand
Wang Computer Ltd.
Auckland
Christchurch
Wellington
Panama
Wang de Panama (CPEC) S.A.
Panama City
Puerto Rico
Wang Computadoras, Inc.
Hato Rey
Singapore
Wang Computer (Pte) Ltd.
Singapore
Sweden
Wang Skandinaviska AB
Stockholm
Gothenburg
Malmö
Switzerland
Wang A.G.
Zurich
Basel
Bern
Geneva
Lausanne
St. Gallen
Wang Trading A.G.
Zug
West Germany
Wang Deutschland, GmbH
Frankfurt
Berlin
Cologne
Düsseldorf
Essen
Freiburg
Hamburg
Hannover
Kassel
Mannheim
Munich
Nürnberg
Saarbrücken
Stuttgart

WANG

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
700-4081G
CR1-3M