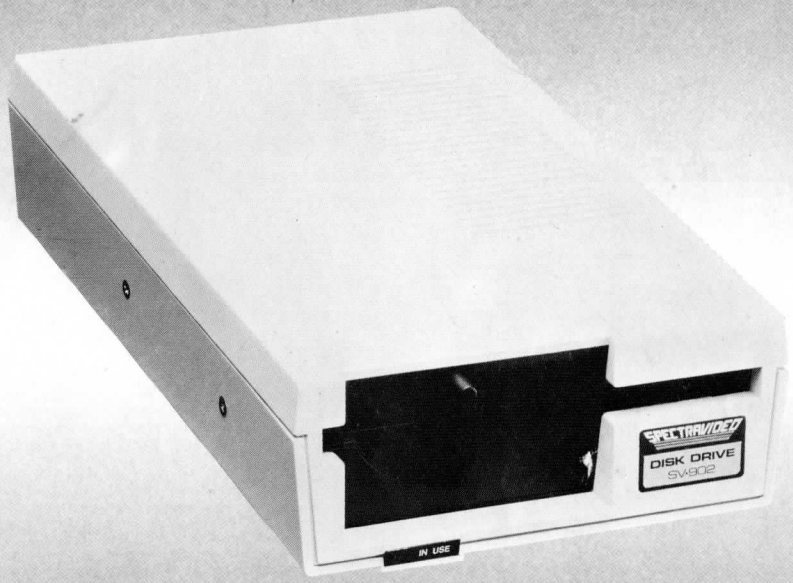
SV·902

DISK BASIC
USER'S GUIDE



SPECTRAVIDEO™

D. Stanley
77 Wilson ST
NEWTOWN
FLAT 3.

Published by
SPECTRAVIDEO INTERNATIONAL LTD.

First Edition
First Printing 1983
Printed in Hong Kong

SV-902 UM/02

CONTENTS

VERY IMPORTANT

Before starting, we recommend that you "write protect" the original BASIC MASTER DISK. "Write protect" ensures that you do not accidentally write something on the BASIC MASTER DISK or erase the vital information that it contains. The "write protect notch" is on the right side of all diskettes. To protect the disk, simply place the "write protect tabs" (that are supplied in the box of diskettes from your local computer dealer) over the "write protect notch."

D. Stanley.

DISK BASIC
USER'S GUIDE

Introduction

Congratulations! The Spectravideo disk drive is your entrance into an expanding universe of computer software and programming tools. The disk drive is light years ahead of the data cassette recorder in speed and power. Where the cassette recorder was slow and inaccurate, the disk drive is fast and secure. This manual is your guide into this new and exciting world. In the coming chapters we will show you how to store and recall your BASIC programs, and how to create data files on floppy diskettes. Please read this manual carefully and try all the programs that we provide as examples.

In theory, the 5-1/4 inch floppy disks (we will use the word "disk" and "diskettes" interchangably) that you have purchased from your computer dealer are manufactured so that they can be used with any microcomputer. However, each microcomputer manufacturer designs his own method or "format" to store information (data) on a disk. That is why a program written for one machine is not necessarily usable on another machine.

Spectravideo, like all other microcomputer manufacturers, requires a disk to undergo a process called formatting to prepare the disk to accept information (sent from the SV-318 or SV-328 computer to the disk drive). This is a simple and quick procedure and it is described below. Be careful to type the commands exactly as they are written.

1. Getting Started— Formatting A Disk

1. Insert into your disk drive the diskette marked **"DISK BASIC"** (we will call this disk the **BASIC MASTER DISK** for the remainder of this guide). If more than one drive is hooked up, use drive number 1.

2. Turn on the Super Expander.

3. Turn on your SV computer.

4. When the computer displays the "OK" message, type:

   Bload "1: svfrmt", r ENTER

   This will load and run the physical format program to format ("initialize" in computerese) the diskette. The computer will then ask you:

   which drive to format?
   1 or 2 or C to abort:

   If you have two disk drives, press the number **2** key. You will then be told to insert the blank disk to be formatted into drive 2 and press **ENTER** . Please do so. If you have only one drive, then press the number **1** key. You will then be told to place your blank disk to be formatted into disk 1 and press **ENTER** . Please do so. After you have pressed the **ENTER** key, the computer will format the disk, printing on the screen the number of the track that it is up to.

When the computer is finished physically formatting the disks, it will then ask you if you wish to format another disk. If you wish to do so then repeat the procedure described from the beginning of step 4. Otherwise press the **CTRL** key and while holding it down press the **C** key.

5. After you have aborted (stopped) the formatting procedure, the computer will instruct you to insert the **BASIC MASTER DISK** into drive 1 and press any key. After you hit any key the computer will then reboot (restart).

6. When the computer prints its "OK" message on the screen after it has rebooted, you must type the following command to complete the formatting process:

   load "1: format", r ENTER

   The computer will then ask you into which drive you will place the disk to be formatted. Type either 1 or 2, then insert the blank disk into the designated drive and press the **ENTER** key.

7. When the "OK" message reappears, the formatting process is over and control of the computer has been returned to you.

2. Preparing a back-up copy of the BASIC MASTER DISK:

Since the **BASIC MASTER DISK** is the key to unlocking the power of your Spectravideo computer, we strongly recommend that you prepare a back-up copy of the **BASIC MASTER DISK** before you continue working with your computer system. We will explain how to do this step by step.

1. First format a blank (new) diskette as we described in the preceding chapter

2. After the disk has been formatted, remove the disk from drive number 1, turn off the computer and then insert the **BASIC MASTER DISK** into drive number 1. Then turn on the power on the computer.

3. After the Spectravideo logo has flashed by and the computer has displayed its "OK" message type:

   load "1:sysgen.bas",r

   The file named "sysgen.bas" that you just told the computer to load and run will copy the disk operating instructions from the **BASIC MASTER DISK** to your newly formatted disk. The computer will ask you:

   Enter source drive
   1, 2 or ^STOP to quit? ■

The "source" refers to the **BASIC MASTER DISK**. Since the **BASIC MASTER DISK** is in drive number 1, type a number **1** and then press **ENTER**. The computer will ask you:

```
Enter for destination drive
1, 2 or ^STOP to quit? ■
```

The "destination" refers to the drive that contains the newly formatted disk that will become the back-up copy of the **BASIC MASTER DISK**.

The destination drive will also be drive number 1, so press the number **1** and then press **ENTER**.

Insert the BASIC MASTER DISK and press any key when the computer asks you to:

```
Insert source disk
in drive 1 press any key
■
```

The computer will then inform you that it is reading the system tracks from the **BASIC MASTER DISK**, which is still in drive number 1, into the computer's memory. This will take approximately four minutes. When this is completed, you will be instructed to remove the **BASIC MASTER DISK** from drive 1 and told to place the newly formatted disk into drive number 1. Please do so and then press **ENTER**. It will take another four minutes to write the information onto the disk. Upon completion of this process you will have a fully formatted **BASIC MASTER DISK** that can be used to boot (start) the system. All that remains to be done is to copy the original files from the original **BASIC MASTER DISK** to your new back-up copy.

4. Insert the original **BASIC MASTER DISK** into drive number 1 and type

   files [ENTER]

The command "files" will make the computer list the names of all the files that are contained on the disk that is in the drive (we will explain everything you need to know about files very shortly).

Any BASIC program can easily be copied by loading the program from the original disk, removing the original from the drive and inserting the disk you want to copy it onto.

Example:

A. Insert **BASIC MASTER DISK.**
B. Type:

load "1:format" ENTER

C. Remove the **BASIC MASTER DISK** and insert the newly formatted disk.
D. Type:

save"1:format" ENTER

Machine language programs are more complicated to copy. A machine language is listed with an asterisk (*) following the filename. There is only one machine language program on the original **BASIC MASTER DISK.** To copy the **"svfrmt*"** program, follow these steps.

A. Insert the **BASIC MASTER DISK.**
B. Type:

bload "1:svfrmt" ENTER

C. Remove the **BASIC MASTER DISK** and insert the newly formatted disk.
D. Type:

bsave "1:svfrmt", &HC400, &HD500 ENTER

After you have finished copying all the files from the original **BASIC MASTER DISK** to the newly formatted disk, you will have a complete **BASIC MASTER DISK** back-up copy.

IMPORTANT

Before continuing, we highly recommend that you "write protect" the original **BASIC MASTER DISK** and your new back up copy of it. "Write protection" ensures that you do not accidently write something on the BASIC MASTER DISK or erase the vital information that it contains.

The "write protect notch" is on the right side of all diskettes. To protect the disk, simply place the "write protect tabs" (that are supplied in the box of diskettes from your local computer dealer) over the "write protect notch."

3. Naming Files

Before the computer age, anytime someone referred to a filing cabinet a picture of a unit of steel drawers loaded with papers probably came to mind. Well, even after the dawn of the computer era, the concept of a file hasn't changed much. A file is a collection of information, kept somewhere other than inside the computer's memory area, that stores programs. That "somewhere" can be a cassette tape or a floppy disk.

To keep order in a filing system—whether it is a conventional filing cabinet or a floppy disk—one must organize the files into distinct units. If one were to label all the files in a drawer with merely the name "bills", it would be difficult to distinguish between bills that have already been paid and those that are still outstanding. Similarly, if one were to name all the files he saved on a diskette in the same way, it would be difficult to distinguish among the files.

There are two different ways to distinguish files, break them into categories and label them properly. One is called a **"filename"** and the other is called a **"filenumber."** We will now describe the criteria for filenames. In chapter 6 we will explain the rules for filenumbers.

Disk filenames can be a maximum of 6 (six) characters in length with an optional character extension that is preceded by a decimal point. For example:

client.law

If a decimal point appears in a filename after fewer than six characters, the name is blank filled to six characters and the next three characters are the extension. For example, if you typed:

```
car.new
```

the computer will list it as

```
car   .new
```

when you instruct the computer to print the names of the files (more on this shortly).

If the filename is 6 or fewer characters with no decimal point, there is no extension. If the filename is more than 6 characters, BASIC inserts a decimal point after the sixth character and uses the next three characters as an extension. Any additional characters are ignored.

4. Saving and manipulation of BASIC programs with DISK BASIC

DISK BASIC contains several commands that are used differently from the similar commands that are used to set up a filing system on the data cassette recorder. The remainder of this manual will describe these commands and how they are used.

The following is a list of commands that allow you to save and manipulate any BASIC program that you write.

We will use the following short program as our example:

```
10 print "hello"
20 goto 10
```

We will first write the general format (syntax) of the command and then apply it to our sample program.

SAVE

Format: **save "1':filename"**

If we were to call our little sample program "hello," we would then write:

Example: **save "1:hello"**

The number 1 in this command and in all subsequent commands tells the computer which drive contains the disk onto which we want to save the program. If the storage disk was in drive 2 you would write:

save "2:hello"

There are two different forms in which your program can be saved. One is called **"tokenized"** form and the other is called **"ASCII"** form. When you routinely save a file as in our example above, your program or data is compacted and one symbol or "token" represents each BASIC command that you use. When your program is tokenized it takes

up less room on the data cassette or floppy disk than if it was untokenized. Each method has its advantages and disadvantages. As you proceed with this tutorial you will see that certain commands work with one type of file and not another.

As previously mentioned, when you type the "**save**" command as we did with the "hello" program, your program is saved as in tokenized form. You have the option of adding an "**a**" to the end of the command to save it as an ASCII file. For example:

save"1:hello",a

LOAD

Format: **load "1:filename"**

Example: **load "1:hello"**

The load command loads the specified program from the disk into the computer's memory, and deletes the current contents of memory. You have the option of adding the letter "**r**" to run the program immediately after it is loaded (as we did in step 4 of chapter 2 of this guide, which formatted the disk). For example:

load "1:hello",r

RUN

Format: **run "1:filename"**

Example: **run "1:hello"**

The **run** command accomplishes the same thing as: load"1:hello",r. It first loads the program from the disk into memory and then executes it.

MERGE

Format: **merge "1:filename"**

Example: **merge "1:hello"**

The **"merge"** command works differently than the commands outlined above. If our "hello" program was stored as an "ASCII" file, then we can load it into memory. The program line numbers are merged with the line numbers of the program that resided in memory before the "merge" was performed. Merging two programs may be thought of as inserting the program loaded from the disk into the program that currently resides in memory. However, the merger is not without a victim. If a line of the current program in memory has the same line-number as one of the lines of the newly loaded program being merged, then only the line from the newly loaded program is retained. After a **merge** command is issued, the merged program resides in memory and control returns to you.

KILL

kill "1:filename"

As its name implies, the **"kill"** command deletes a file from the disk. To delete our sample program type:

kill "1:hello"

NAME

name"1:old-filename" as "1:new-filename"

The **"name"** command allows you to rename a file. To rename our "hello" program type:

name "1:hello" as "1:greet"

FILES

The **"files"** command displays the names of the files residing on a diskette. If you type

files

the computer will display the files residing on the diskette in drive 1. To display the files on a diskette in drive 2 type:

files2

5. Sequential Data Files

There are two different types of diskette data files that may be created and used by a BASIC program. One is a "sequential file" and the second is a "random access file." Each has its advantages and disadvantages, as discussed below.

Sequential files are easier to create than random files, but are limited in speed and flexibility when it comes to accessing data. As its name implies, the data is written sequentially, that is one item after the other, in the order it is sent to the diskette. It is loaded back into the computer in the same way.

The following steps must be included in a program to create and access a sequential file. The ease of working with data files will be greatly enhanced if you continually bear in mind the parallel between a computer data file and the folder file stored in a filing cabinet. The following 3 steps are common to both.

1. **OPEN** the file for output (from the computer to the disk drive) or appending (adding to it.)

2. **WRITE** data to the file using the PRINT# command (or other commands).

3. **CLOSE** the file after you have written to it. To read data from a file you must OPEN it again for input (from the disk drive into the computer).

Note: The above mentioned commands, and several others that are used to create and access sequential commands, are illustrated and explained through the use of sample programs. Please type the programs carefully and try them out.

DEMO#1

The first demonstration program highlights the following four fundamental commands:

OPEN
CLOSE
PRINT
INPUT

```
10 open "1:demo1" for output as #1
20 a = 10: b = 20
30 c = 30
40 print #1, a;b;c
50 close#1
60 open "1:demo1" for input as #1
70 input#1, a,b,c
80 print a,b,c
90 close#1
```

This program will save the numbers 10, 20, and 30 on the disk then read them and print them on the screen. Here's why:

Line 10 instructs the computer to OPEN (create) a file on drive 1 called demo#1 to which we will output, or write information. The #1 at the end of line 10 is the filenumber for the demo#1 file.

As was explained in chapter 3, every data file is referred to with a filename and filenumber. The filename is the label that you use to refer to the file. The filenumber is what the computer uses to refer to the file. The filenumber is a unique number that is associated with the physical file when it is **open**ed. It identifies the route that the computer uses to send and receive information with the disk drive. Very rarely will you need to access more than one file at a time. Therefore, use filenumber 1 for those instances when you are using only one data file. However, should you wish to open more than one file at a time, you must specify in your program how many files you wish to open. To specify the maximum number of files you will open at once, use the following format:

Format: **maxfiles = [number of files]**

Example: **10 maxfiles = 2**
10 maxfiles = 5

Lines 20 and 30 define the variables the program will use.

Line 40 is the one that actually instructs the computer to write them on the disk, and line 50 closes the demo#1 file (filenumber 1).

On line 60 the computer is instructed to reopen the file to be able to read the information back into the computer. Notice that the filenumber again is #1.

Line 70 causes the computer to read the information back into the computer, and line 80 prints out the specified variables. Line 90 closes the demo#1 file.

DEMO#2

This program illustrates the **LINE INPUT#** command.

```
10 open "1:demo2" for output as #1
20 a$ = "this is a demonstration"
30 b$ = " this is part of it too"
40 print #1, a$,b$
50 close#1
60 open"1:demo2" for input as #1
70 line input#1, a$
80 close#1
```

This program writes the message contained on lines 20 and 30 on the disk, then reads it back and prints it on the screen. The new command **line input#** appears on line 70. This command reads an entire line (up to 254 characters), without delimeters, from a sequential file to a string variable.

DEMO#3

This program demonstrates how to **append** new information to an existing sequential file.

```
10 open "1:demo 3" for output as #1
20 a$ = "this is a demonstration"
30 b$ = " this is part of it too"
40 print#1, a$,b$
50 close#1
60 c$ = "so is this"
70 open"1:demo3" for append as #1
80 print#1,c$
90 close#1
100 open "1:demo3" for input as #1
110 line input#1, d1$
120 line input #1, c1$
130 print d1$: print c1$
140 close#1
```

Lines 10-50 are the same as those in the demo#2 program above. Lines 70-90 reopen the demo#3 file and write the message contained in c$. Then lines 100-140 open data file demo# 3, then read in d1$ (which consist of a$ and b$) and c1$ (which consists of c$) and then print d1$ and c1$.

DEMO#4

This program demonstrates the last major command needed for sequential data file creation and access. The command is **EOF,** which is the abbreviation for **"End Of File."**

```
10 open"1:demo4" for output as #1
20 for a = 0 to 50
30 print#1,a
40 next a
50 close#1
60 open"1:demo4" for input as #1
70 if eof (1) then goto 120
80 input#1,a
90 print a
100 goto 70
110 close#1
120 print "all done"
```

This program writes the numbers 0-50 into a file and then reads them back and prints them on the screen. It prints the message "all done" when it finishes. What is so great about that

you ask? Well, before we explain the function of EOF, delete line 70 from the program, change line 100 to read "goto 80" and then **run** the program. Did you get this error message?:

```
input past end in 80
```

You probably did, because after the computer prints the last item in the file-number 50-it returns to the file looking for more data to read because line 100 sent it to line 80 which tells it to read. But since there is no more data left in the file, you are told that you tried to input (transfer from disk to computer) past the end of the file.

So how does the EOF command help? The EOF function tests to see whether or not the end of a file is reached. If the end of a file has been reached (true) then the value that EOF returns (transmits) to the program is 1 (one). A 0 (zero) will be returned if the end of the file has not been reached. Now let's look at line 70 again. Here is how to read it:

If the end of the file has been reached, then goto 120. Before each item is read, the EOF tests to see if the end of file has been reached. If it has not been reached (the false or zero condition), the program continues to line 80. However if the EOF test reports a true (1) condition then the program jumps to line 120 and prints the "all done" message rather than the "input past end" error message.

This brings to a close our discussion of sequential data files. Several minor commands that can be used when working with sequential data files have not been demonstrated. They are:

PRINT# USING **LOC**
INPUT$ **LOF**

These commands are described in a valuable reference work, Spectravideo's "BASIC Reference Manual." It can be bought where Spectravideo products are sold.

6. Random Access Files

Creating and accessing random files requires more programming steps than is the case with sequential files, but there are advantages to using random access files. Random files are stored in the tokenized format while a sequential file is stored as ASCII characters.

The biggest advantage of random files is that data can be accessed anywhere on the diskette (randomly). This means that, unlike sequential files, it is not necessary to read through all the files one after another until the file you desire is found. This is so because the information that comprises a random file is stored and accessed in distinct units called **"records,"** and each record is numbered.

The following programming steps are required to create a random file.

1. **OPEN** a file for random access.

2. The data must first be moved from the program area of memory to a random buffer prior to writing it on a disk. The **FIELD** command allocates space for the data in the random buffer.

3. Use the **LSET** or **RSET** commands to position the data in the random buffer.

4. Write the data from the buffer to the diskette using the **PUT** statement. You need not close a random file before accessing (reading) the information back into the computer (as was the case with sequential files).

The following programming steps are required to access a random file.

1. **OPEN** a random file, if it was previously closed.

2. Use the **FIELD** statement to allocate space in the random buffer, if the file was previously closed.

3. Use the **GET** command to move the desired record into the random buffer.

You need not close a random file before you read from it. Therefore, we will write the following demonstration program in two different ways. The first way will open, write and close the random file, then reopen it, read it, print it and close it. The second way opens, writes, reads, prints and closes it.

DEMO#5

```
 10 input "customer name:";q$
 20 input "city:";r$
 30 open"1:demo5" as #1
 40 field #1, 20 as n$, 10 as a$
 50 lset n$ = q$
 60 lset a$ = r$
 70 put#1,18
 80 close#1
 90 open"1:demo5" as 31
100 field #1, 20 as n$, as a$
110 get#1,18
120 print n$: print a$
130 close#1
```

This program is the beginning of a database to hold customer names and their cities. We could have written it as:

```
 10 input "customer name:";q$
 20 input "city:";r$
 30 open"1:demo5" as #1
 40 field #1, 20 as n$, 10 as a$
 50 lset n$ = q$
 60 lset a$ = r$
 70 put#1,18
 80 get#1, 18
 90 print n$: print a$
100 close #1
```

Here is how the program works:

Lines 10 and 20 store the customer information in strings q$ and r$. Line 30 opens demo#5. Line 40 allocates the space for the information about the customers in a random buffer. It allocates 20 positions (bytes) for n$, and 10 positions for a$. n$ and a$ are the string variables in the string buffer that will hold the information about the customers that was originally in q$ and r$.

The **LSET** commands in lines 50 and 60 move the data from the q$ and r$ variables and places it into the string variables, n$ and a$ which are in the random buffer. Line 70 writes the record (the data) from the random buffer to the data file. The number 18 is the number of the record that we have arbitrarily chosen. You should be careful when you number your records because organization is the key to moving the data around among the program area, the random buffer and the random file. The **GET** command reads the data back into the random buffer from a random file.

The **LSET** command justifies the string variable to the left, and the **RSET** command justifies it to the right. (For a more complete description of the **LSET** and **RSET** commands, see the BASIC Reference Manual.)

DEMO#6

Our previous program (demo#5) used only string variables. However, there will probably be many situations where you need to store numerical information in a random access file too. Before doing so, you must add on two extra programming steps. The first step converts a numeric type value into a string type value before you write the data to the diskette. The second extra step converts the string variable type back into its numeric value. The following program demonstrates two of the commands that perform this conversion.

```
 10 input "customer name:"; cust$
 20 input "city:"; city$
 30 input "phone number:"; tel
 40 open "1:demo6" as 31
 50 field #1, 20 as n$, 10 as a$, 8 as t$
 60 lset n$ = cust$
 70 lset a$ = city$
 80 lset t$ = mkd$(tel)
 90 put #1, 18
100 get #1, 18
110 t = cvd(t$)
120 print n$: print a$: print t
```

This program writes the customer's name, city and telephone number on the disk, reads it back, and prints it on the screen. The new commands introduced in this program are on lines 80 and 110. Line 80 uses the **MKD$** command to convert the numeric data stored in "tel" into a string variable called t$. This allows the telephone number to be written to the disk along with the other customer information which was typed in string form by the user. Later, after the information from the random file has been read, the **CVD** command converts the string variable t$ into a numeric value which is stored in "t."

This brings to a close our discussion of random access file commands. We have not described several minor commands. They are:

CVI
CVS
LOC
LOF
MKI$
MKS$

These commands are described in the BASIC Reference Manual.

APPENDIX A

DISK BASIC ERROR MESSAGES

Field overflow — A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

Internal error — An internal malfuction has occurred in DISK BASIC. Report to Microsoft the conditions under which the message appeared.

Bad file number — A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

File not found — A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

File already open — A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

Disk I/O error — An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

File already exists — The filename specified in a NAME statement is identical to a filename already in use on the disk.

Disk full — A disk storage space is in use.

Input past end — An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

Bad record number — In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

Bad file name

An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

Direct statement in file

A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

Too many files

An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

APPENDIX B

TECHNICAL INFORMATION FOR THE EXPERIENCED USER

A. Drive Information

For each disk drive that is **mounted**, the following information is kept in memory:

1. Attributes

Drive attributes are read from the information sector when the drive is mounted, and may be changed with the SET statement. Current attributes may be examined with the ATTR$ function.

2. Track Number

This is the current track while the disk is mounted. Otherwise, track number contains 255 as a flag that the disk is not mounted.

3. Modification Counter

This counter is incremented whenever an entry in the File Allocation Table is changed. After a given number of changes has been made, the File Allocation Table is written to disk.

4. Number of Free Clusters

This is calculated when the drive is mounted, and updated whenever a file is deleted or a cluster is allocated.

5. File Allocation Table

The File Allocation Table has a one-byte entry for every cluster allocated on the disk. If the cluster is free, this entry is 255. If the cluster is the last one of the file, this entry is 300 (octal) plus the number of sectors that were used from this cluster. Otherwise, the entry is a pointer to the next cluster of the file. The File Allocation Table is read into memory when the drive is mounted, and updated:

5.1. When a file is deleted

5.2. When a file is closed

5.3. When modifications to the table total twice the number of sectors in a cluster (this can be changed in custom versions).

5.4. When modifications to the table have been made and the disk head is on (or passes) the directory track.

B. Directory Format

On the diskette, each sector of the directory track contains eight file entries. Each file entry is 16 bytes long and formatted as follows:

Bytes	Usage
0-8	Filename, 1 to 9 characters. The first character may not be 0 or 255.
9	Attributed: Octal 200 Binary file 100 Force read after write check 20 write protected file Excluding 200, these bits are the same for the disk attributed byte which is the first byte of the information sector.
10	Pointer into File Allocation Table to the first cluster of the file's cluster chain.
11-15	Reserved for future expansion.

If the first byte of a filename is zero, that file entry slot is free. If the first byte is 255, that slot is the last occupied slot in the directory, i.e., this flags the end of the directory.

C. File Block

Each file on the disk has a file block that contains the following information:

1. **File Mode** (byte 0)
This is the first byte (byte 0) of the file block, and its location may be read with VARPTR (#filenumber). The location of any other byte in the file block is relative to the file mode byte. The file mode byte is one of the following:

(octal)

1	Input only
2	Output only
4	File mode
10	Append mode
20	Delete file
200	Binary save

NOTE

It is not recommended that the user attempts to modify the next four bytes of the File Allocation Table. Many unforeseen complications may result.

2. Pointer to the File Allocation Table entry for the first cluster allocated to the file (+ 1)

3. Pointer to the File Allocation Table entry last cluster accessed (+ 2)

4. Last sector accessed (+ 3)

5. Disk number of file (+ 4)

6. The size of the last buffer read (+ 5). This is 128 unless the last sector of the file is not full (i.e., Control-Z).

7. This current position in the buffer (+ 6). This is the offset within the buffer for the next print or input.

8. File flag (+ 7), is one of the following:

octal	
100	Read after write check
20	File write protected
10	Buffer changed by PRINT
4	PUT has been done. PRINT/INPUT are errors until a GET is done.
2	Flags buffer is empty

9. Terminal position for **TAB** function and comma in PRINT statements (+ 8).

10. Beginning of sector buffer (+ 9), 128 bytes in length.

D. Advanced Uses of File Buffers

1. Information may be passed from one program to another by FIELDING it to an unopened file number (not #0). The **FIELD** buffer is not cleared as long as the file is not OPENed.

2. The FIELDed buffer for an unopened file can also be used to format strings. For example, an 80-character string could be placed into a FIELDed buffer with LSET. The strings could then be accessed as four 20-character strings using their FIELDed variable names. For example:

```
100  FIELD#1, 80 AS A$
200  FIELD #1, 20 AS A1$, 20 AS A2$,
     20 AS A3$, 20 AS A4$
300  LINE INPUT "CUSTOMER
     INFORMATION: ":B$
400  LSET A$ = B$
500  PRINT "NAME "; A1$,"SSN: ";A2$
```

3. **FIELD#0** may be used as a temporary buffer, but note that this buffer is cleared after each of the following commands: **FILES, LOAD, SAVE, MERGE, RUN, DSK$, OPEN.**

E. Disk Allocation

With SV Super Extended Disk BASIC, storage space on the diskette is allocated beginning with the cluster closest to the current position of the head. (This method is optimized for writing. Custom versions can be optimized for reading.) Disk allocation information is placed in memory when the disk is mounted and is periodically written back to the disk. Because this allocation information is kept in memory, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files.

F. Double Density

The **DSKI$** function returns as its value the first 255 bytes of the sector read.

The **DSKO$** statement does not use the string expression field. The format is:

DSKO$ drive , track , sector

In order to specify the data to write with DSKO$ and to retrieve all 256 bytes of the data read by DSKI$, the user must FIELD two or more variables (for a total of 256 bytes) to the file#0 buffer. The FIELDed variables will be identical to the data read with DSKI$ and written with DSKO$. For example:

FIELD#0,128 AS A$, 128 AS B$

G. Filenames

The format for disk filenames is:

drive #: filename. extension

The first drive is 1.

Disk filenames are six characters with an optional three-character extension that is preceded by a decimal point. If a decimal point appears in a filename after fewer than six characters, the name is blank-filled to six characters and the next three characters are the extension. If the filename is six or fewer characters with no decimal point, there is no extension. If the filename is more than six characters, BASIC inserts a decimal point after the sixth character and uses the next three characters as an extension. (Any additional characters are ignored.)

H. File Format

Each file requires 137 bytes: 9 bytes plus the 128-byte buffer. Because the File Allocation Table keeps random access information for all files, random and sequential files are identical on the disk. The only distinction is that sequential files have a Control-z (32 octal) as the last character of the last sector. When this sector is read, it is scanned from the end for a non-zero byte. If this byte is Control-z, the size of the buffer is set so that a PRINT overwrites this byte. If the byte is not Control-z, the size is set so the last null seen is overwritten.

Any sequential file can be copied in random mode and remain identical. If a file is written to disk in random mode (i.e., with PUT instead of PRINT) and then read in sequential mode, it will still have proper end of file detection.

I. Disk Files

The **FILES** command points the names of the files residing in a disk. The format is:

[L] FILES [< drive number >]

LFILES outputs to the line printer. In addition to the filename, the size of each file, in clusters, is output. A cluster is the minimum unit of allocation for a file—it is one-half of a track.

Filenames of files created with OPEN or ASCII SAVE are listed with a space between the name and extension. Filenames of binary files created with binary SAVE are listed with a decimal point between the name and extension.

Files created by the SAVE filenames command to save the current screen image are listed with a pound sign (#) between the name and the extension. The protected file option with SAVE is not supported in SV Super Extended Disk BASIC.

J. Open Statement

The format for the **OPEN** statement in SV Super Extended BASIC is:

OPEN <filename> [FOR <mode >] AS [#] < file number >

where <mode> is one of the following:

INPUT
OUTPUT
APPEND

The mode determines only the initial positioning within the file and the actions to be taken if the file does not exist. The action taken in each mode is:

INPUT The initial position is at the start of the file. An error is returned if the file is not found.

OUTPUT The initial position is at the start of the file. A new file is always created.

APPEND The initial position is at the end of the file. An error is returned if the file is not found.

If the **FOR** <mode> clause is omitted, the initial position is at the start of the file. If the file is not found, it is created.

Note that variable length records are not supported in SV Super Extended Disk BASIC. All records are 128 bytes in length.

When a file is **OPENed for APPEND,** the file mode is set to **APPEND** and the record number is set to the last record of the file. The program may subsequently execute disk I/O statements that move the pointer elsewhere in the file. When the last record is read, the file mode is reset to FILE and the pointer is left at the end of the file. Then, if you wish to append another record, execute:

```
GET#n, LOF(n)
```

This positions the pointer at the end of the file in preparation for appending.

At any one time, it is possible to have a particular filename OPEN under more than one file number. This allows different attributes to be used for different purposes. Or, for program clarity, you may wish to use different file numbers for different methods of access. Each file number has a different buffer, so changes made under one file are not accessible to (or affected by) the other numbers until the record is written (e.g., GET#n, LOC(n).

K. SET Statement

The **SET** statement determines the attributes of the currently mounted disk drive, a currently open file, or a file that need not be open. The format of the SET statement is:

SET < drive> , # < file>
< filename>,<attribute string>

An "attribute string" is a string of characters that determines what attributes are set. Any character other than the followings are ignored:

R Read after write
P Write protect

Attributes are assigned in the following order:

1. SET <drive> ,<attribute string> Statement
   This statement sets the current attributes for the disk. The attributes are permanently recorded.

2. When a file is created, the permanent file attributes recorded on the disk will be the same as the current drive attribute.

3. SET <filename>,<attribute string> Statement
   This statement changes the permanent file attributes that are stored in the directory entry for that file. It does not affect the drive attributes.

4. When an existing file is OPENed, the attributes of the file number are those of the directory entry.

5. SET#<file number >,< attribute string >Statement
   This statement changes the attributes for that file number but does not change the directory entry.

Examples:

SET 1, "R"
Force read after write checking on all output to drive 1.

SET #1, "R"
Force read after write for all output to file#1 while it is open

SET #1, "P"
Give write protect error if any output is attempted to file 1

SET "drive#: TEST", "P"
Protect TEST from deletion and modification

SET 1, " "
Turn off all attributes for drive 1

L. FORMAT Program

Before mounting a drive with a new diskette, run BASIC's FORMAT program to initialize the directory (set all bytes to 255), set the information sector to 0, and set all the File Allocation table entries (except the directory track entry (254) to "free" (255).

The FORMAT program is:

```
10  '
20  '  This Program formats a diskette
30  '  to be used in disk BASIC.
40  '  Note that this is only a logical
50  '  formatter. I.e. clears directory
60  '  and file allocation table.
70  '  Physical formating should be
80  '  done by CP/M's FORMAT Program
90  '
100 '
110 '
120 '
130     CLEAR 800
140     INPUT "Which drive";D
150     FIELD #0, 128 AS A$, 128 AS B$
160 '
170 '  Clear directory entry
180 '
190     LSET A$ = STRING$ (128,255)
200     LSET B$ = STRING$ (128,255)
```

```
210     FOR S = 1 TO 13
220     DSKO$ D,20,S
230     NEXT S
240     '
250     '  Clear Disk allocation table
260     '
270     LSET A$ = STRING$ (128,0)
280     LSET B$ = STRING$ (128,0)
290     DSKO$ D,20,14
300     '
310     '  Clear file allocation table
320     '
330     LSET A$ = STRING$ (3,254) +
        STRING$ (17,255) + CHR$ (254) +
        STRING$ (19,255)
340     DSKO$ D,20,15
350     DSKO$ D,20,16
360     DSKO$ D,20,17
317     GOTO 140
```

After running FORMAT files will be allocated as usual, i.e., on either side of the directory track.

M. Advanced Commands

FPOS

The **FPOS** function:

FPOS (< file number >)

FPOS returns the number of the physical sector where **<** filenumber **>** is located.

DSKO$

The **DSKO$** statement:

DSKO$ **<** drive **>** ,**<** track **>** ,**<** sector **>** , **<**string expression**>**

writes the string on the specified sector. The maximum length for the string is 256 characters. A string of fewer than 256 characters is zero-filled at the end to 256 characters.

DSKI$

**DSKI$** is the complementary function to the DSKO$ statement. DSKI$ returns the contents of a sector to a string variable name.

IPL

The IPL command instructs Disk BASIC to immediately execute the program you select when this disk is booted.

Format: **IPL "RUN" + chr$(34)+ "1;filename"**

DSKI$ (<drive> ,<track> ,<sector>)

DISK I/O

GET — If the "buffer changed" flag is set, write the buffer to disk. Then execute the GET (read the record into the buffer), and reset the position for sequential I/O to the beginning of the buffer.

PUT — Execute the PUT (write the buffer to the specified record number), and set the "sequential I/O is illegal" flag until a GET is done.

INPUT# — If the buffer is empty, write it if the "buffer changed" flag is set, then read the next buffer.

PRINT # — Set the "buffer changed" flag. If the buffer is full, write it to disk. Then, if the end of file has not been reached, read the next buffer.

ATTR$ FUNCTION

**ATTR$** returns a string of the current attributes for a drive, currently open file, or file that need not be open. The format of ATTR$ is:

ATTR$ (<drive> #<file number> , <filename>)

Example:

```
SET 1, "R":A$#ATTR$ (1):PRINT A$, R
Ok
```

APPENDIX C

FOR USERS THAT OWN A 64K RAM EXPANSION CARTRIDGE

To create a back up copy of your SV Extended Disk BASIC diskette, follow these steps:

1. Initialize a new diskette for use with the system as described in chapter 2.

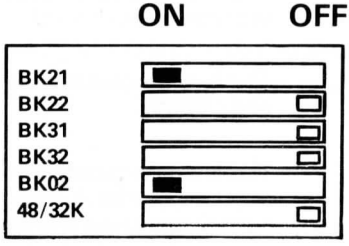
2. Change the switches on your 64K card as shown in Figure 1. This change is necessary to run CP/M.

3. Insert the CP/M diskette into drive 1 and turn the power on for both the expansion box and the SV-318 or SV-328 computer.

4. Type **COPY** and press the **ENTER** key when the prompt appears. The COPY program will tell you to put your diskette into drive A (the original disk BASIC diskette) and the disk that is to receive the copy in drive B (the newly formatted diskette).

When the computer is through copying, you will have a back up copy of your BASIC Master diskette.

Note: If only one disk drive is available, type **1COPY** (instead of COPY) and follow the instructions that are provided.



NORMAL CONFIGURATION TO RUN CP/M BANK 2 Page 1 and BANK 0 Page 2

Figure 1

APPENDIX D

CONTENTS OF THE SV EXTENDED DISK BASIC DISKETTE

This diskette contains a number of useful, entertaining and teaching programs. They can be used for the utility functions that some of them are designed for or may be used as programming aids. They can also be used to demonstrate your system to friends and family.

The programs contained on the diskette are as follows:

svfrmt*

This allows you to format a blank diskette for use as either a CP/M diskette or an SV Extended Disk Basic Diskette. This must be done prior to using a blank diskette in either system. For use as an SV Extended Disk Basic diskette, this operation must be followed by running the FORMAT program listed below.

sysgen. bas

This allows you to create a diskette that will boot (or execute when the system is first turned on) and contains all of the attributes of SV Extended Disk Basic.

dsktst

This is a diagnostic program that will test the diskette itself to insure that it has no bad sectors or other problems that might cause a loss of data. It is a good idea to run this program anytime you initialize a new diskette.

demo 1

This a demonstration of the graphics capability of your computer.

demo 2

This is a demonstration of sprites, music, color and graphics.

demo 3

This is a demonstration of the sound capabilities of your computer. It contains a melody that has been programmed entirely in Basic.

demo 4

This is a demonstration of the colors that are available on your computer.

init

This allows you to select a program that is contained on the diskette to automatically run whenever the system is turned on and that diskette is in Drive 1.

format

This allows you to format a previously prepared (using SVFRMT) diskette for use as an SV Extended Basic diskette. To make this diskette into an SV Extended Disk Basic System Master Diskette, you must now run the SYSGEN program listed below.

plist

This allows you to maintain a simple listing of phone numbers that you wish to keep track of.

SP .dat

This is the data file for the SPRITE.DEM program. It cannot be run by itself because it does not contain BASIC statements.

ipl

This program will redefine special function keys as follows:

Key 1 — Files
Key 2 — load "1:
Key 3 — Save "1:
Key 4 — List
Key 5 — Run
Key 6 — Colour
Key 7 — Cload
Key 8 — Cont
Key 9 — List
Key 10 — Run

Keys 1, 2 and 3 are commands for disk operations only.

This program automatically runs when disk is booted. If you run a different program and you wish to redefine the keys to the above configuration just type "RUN" 1:ipl"

SV 902 UM/02

Printed in Hong Kong