HINTS FOR NEW DISK DRIVE OWNERS

FROM

THE COCONUTS

TANDY TRS 8Ø COLOUR COMPUTER

HINTS FOR NEW DISK DRIVE OWNERS

Compiled by Bob Devries and Graham Butcher.

Your new disk drive will be one of your greatest assets after the COCO. The new slimline and other models are far superior to the Tandy drive and controller. As they are not Tandy, a few things should be pointed out.

The controllers and drives have been set up to be totally compatible with disks written on Tandy drives set to 35 tracks. All comments for the Tandy drives are the same on your drive.

You have probably spent some time deciding on your purchase, time waiting for it to be set up, and now want to start using it.

SETTING UP

Firstly make sure that all power is turned off. Insert the controller into the ROM port and connect the cable. Power up all accessories before the COCO. Never remove the controller from the computer with the power on, as serious damage could be caused.

When you power up the COCO you should be greeted with the name and version of the rom installed in your controller eg. DISK EXTENDED COLOR BASIC etc. This tells you that the disk basic contained in the controller is being used. This message will always appear on the screen on power up with the controller plugged in even with no power to the drive.

THE DISKETTES

The floppy disk or diskette comes in several types. Only soft sectored, double density disks may be used on the COCO so be careful when buying yours. The disks need to be initialised or prepared to be used to store programs or data. Your first command to your new drive will be one to initialise a disk.

Make sure the disk is in the drive the right way. Hold the disk by the top and take it from its protective envelope. Most diskettes have a label on them and all have a small section about 5mm square cut from them. On the new slimline drives which rest horizontally, the cut notch is on the left of the drive door keeping the label up. On the Tandy drives which rest vertically , the notch is to the top and the label on the right hand side. Never touch any part of the disk which can be seen through the slot as this will damage the diskette. Treat the diskettes with care. They are reasonably priced these days but the files they contain may represent many long hours of your work and therefore could be difficult to replace.

Insert the disk into the drive and close the door. Different brands of drives have different types of door closings. Check to make sure you know how your drive door closes. You will always get an I/O if the door is open and you try to access your disk.

PREPARING THE DISKETTE

Now with the disk inserted type DSKINIØ and press <ENTER>. The drive will begin working to initialise the diskette in drive Ø and take about 3Ø seconds before you get an OK prompt. Now type DIR and press <ENTER>. You will get another OK prompt. Your command DIR asked the computer to look at track 17 and print on the screen the names of all programs contained on the disk. As you have just initialised the disk there are no programs on it hence the directory is empty.

If you have a double sided drive, you may store programs on the other side of the disk without removing it from the drive. Your drives are set up as drive Ø and drive 2. When you typed in DIR it was the same as typing in DIRØ. Try typing in DIR2. The drive will work for a while and come back with an I/O error. This is because the disk has not been initialised and cannot be read. Now you can initialise this second side with the command DSKINI2 and press <ENTER>. At the OK prompt type DIR2 and you will see the OK prompt as you saw at DIR (DIRØ).

STORING FILES ON THE DISKETTE

Saving programs to disk is the same as saving to tape except the C in CSAVE is removed to leave a command SAVE"PROGRAM"for a basic program and by SAVEM"PROGRAM",start address,end address, exec address for a ML program. Load programs from disk with the command LOAD"PROGRAM" for basic and LOADM"PROGRAM" for ML programs.

There are several programs available that will transfer programs from tape to disk but try it this way for a while. It will help you to understand your system better. Save some programs onto your initialised disk and we will look at the directory.

DISK FREE SPACE

Finished? You may have got carried away and got a DF error. This means that the disk is full. You may check to see if there is any free space on a disk with the command ?FREE(Ø). The disk will whirl for a while and a number between Ø and 68 will be printed on the screen. This tells you how many free "GRANULES" there are on that side of the disk for you to store programs. 68 means that the disk is empty and Ø means that the disk is full. There may be a few granules free on a disk and yet when you try to save a program onto it you get a DF error. This means that the program you are trying to save needs more space than is left. If you have a double sided drive, check the free granules on drive 2 with the command ?FREE(2).

THE DIRECTORY

Let's look at the directory. Type DIR and press <ENTER>. You will see letters and numbers set out in five columns. The first column gives the names of the programs while the second column indicates the type of extension the program has. If you saved a basic program with SAVE"PROGRAM" the computer will assign the extension BAS. If you saved a ML program with SAVEM"PROGRAM",start address,end address,exec address then the computer will assign the extension BIN. It is possible to assign your own extension eg SAVEM"PROGRAM/XXX",start,end, exec. Files from Database programs will have the extension DAT.

The next three columns store technical information which are not required for normal use but will be briefly explained.

The third column will have a number between Ø and 3.

Ø	Basic program
1	Data created by a basic program
2	Data created by a ML program
3	A source program created by an editor/assembler

The fourth column lists the format the file is stored in.

A	Ascii
B	Binary

The fifth column shows how many granules the file uses.

If you have stored a large number of files on a disk they will flash by, leaving you with only the last 14 shown on the screen. You may stop this, but it depends on the type of disk rom installed in your controller. The Tandy version will stop scrolling if you press the <SHIFT> and <@> at the same time but you need to be quick. Other versions will stop at the time when the screen is full. Continue scrolling by pressing any key.

BACKUP

Some disks may contain programs you feel warrant a second or backup copy. You use the BACKUP command to do this.

If you have only one drive, initialise your backup disk and then insert your program disk (source diskette). Type BACKUPØ and press <ENTER>. The drive will work for a few seconds and then prompt you to "INSERT DESTINATION DISKETTE AND PRESS ENTER". The drive will write to the new disk and then prompt you to "INSERT SOURCE DISK AND PRESS ENTER". Continue until the OK prompt tells you that the BACKUP is finished. You will notice that it takes longer to write the destination disk than it takes to read the source disk. This is because while the drive is writing to the destination disk it is also verifying that the data was written correctly.

COPYING A FILE

Programs may be transferred from disk to disk by using the command COPY. This command takes the whole file and rewrites it to another disk or another side of the same disk without you having to worry about addresses in ML programs.

Copying from one side of a disk to the other side of the same disk is done using the command line as set out here.

```
COPY"PROGRAM/EXT:Ø" TO "PROGRAM/EXT:2" and press <ENTER>
```

Substitute the name and extension of your program into the command line above.

```
COPY"LOGO/BIN:Ø" TO "LOGO/BIN:2" OR

COPY"MATHTEST/BAS:Ø" TO "MATHTEST/BAS:2"
```

You can also use the COPY command to transfer the program from one disk to the another. Place the SOURCE disk in your drive and type:

```
COPY"PROGRAM/EXT:Ø" and press <ENTER>
```

Then follow the screen instructions until the file has been transferred.

KILLING A FILE

You may decide to remove a file from your disk permanently. This is done using the KILL command. Place your disk in the drive and type in DIR. Check the correct spelling and extension of the file and then type in:

```
KILL"PROGRAM/EXT:DRIVE NUMBER"
```

To completely remove all files from a disk, use the initialising command DSKINI.

This first part of the booklet is just to get you started using your new drive system. You are now ready to start PART II.

Disk Extended Basic Instruction Set.
An in-depth view.

In this section the DISK EXTENDED COLOR BASIC instruction. set will be explained in some detail. For quick reference please turn to the last section of these notes.

The format used in these explanations will be as follows:-

INSTRUCTION NAME

This is the name that the command is known by, e.g. DIR

SYNTAX

The syntax section shows the correct format for the instruction as well as all parameters, whether required or optional.

PURPOSE

This section explains what the instruction is used for, and provides some background to help you understand how and when to use it.

EXAMPLE

This section provides one or more examples which illustrate typical applications of the instruction.

A number of key words, abbreviations, and symbols are used throughout this manual to indicate parameters. They are :-

SYMBOL	MEANING	EXAMPLES
filename	Name of program	"MYPROG"
/ext	filename extension	/BAS, /DAT, /BIN
:d	drive number	:Ø, :2
#buffer	buffer number	#1, #5
data	constant or variable	A, 1, "STRING"

Most of the instructions in DISK BASIC include parameters which modify the operation of the instruction in some way. The following notation conventions are used to describe these parameters :

<parameter>

Parameters which MUST be included with an instruction are shown in angle brackets "<>". The angle brackets are not included in the instruction, but everything inside them is.

[parameter]

Parameters which MAY be included with an instruction are shown in square brackets "[]". The square brackets are not included in the instruction, but everything inside them is.

",..."

A comma followed by three periods is used to indicate the optional continuation of a list. This indicates that the immediately preceding parameter may be continued to a maximum of 24Ø characters.

SYNTAX :

```
     BACKUP <source drive> TO <destination drive>
```

or

```
     BACKUP <drive>
```

PURPOSE :

The BACKUP command is used to make a copy of a disk. The disk to be copied is referred to as the "source disk", since it is the source of the data to be copied. The disk which is to be the copy is referred to as the "destination disk", since it is the destination of the data to be copied. The BACKUP command will copy the entire contents of the source disk to the destination disk.

The BACKUP command will copy the entire contents of the source disk, sector-by-sector, to the destination disk. If there is any data on the destination disk, it will be written over. The destination disk must be formatted (see DISKINI).

The action taken by the BACKUP command will be different for single drive BACKUPs than for multiple drive BACKUPs.

USING THE BACKUP COMMAND ON MULTIPLE-DRIVE SYSTEMS :

EXAMPLE :

```
     BACKUP Ø TO 1
```

The computer will read data from the disk in drive Ø and write it to the disk in drive 1. This procedure will continue until the computer has read all of the sectors of the disk in drive Ø, and written all of the data to the disk in drive 1.

WARNING : The BACKUP command will destroy data or programs in memory! DO NOT use this command if you have a program in memory that you want to keep! Use the SAVE command to save the program prior to using the BACKUP command.

USING THE BACKUP COMMAND ON SINGLE-DRIVE SYSTEMS :

EXAMPLE :

```
BACKUP Ø
```

This is the only valid syntax for the BACKUP command on systems with only one drive. Place the source disk in the disk drive, and enter the command "BACKUP Ø". Since the disk contains more data than your computer can hold in it's memory, you will have to help! First, the computer will read some data from the source disk into it's memory. Then, the computer will instruct you to insert the destination disk in the disk drive:

```
INSERT DESTINATION DISKETTE AND PRESS ENTER
```

Remove the source disk and insert the destination disk in the drive, close the door, and press the "ENTER" key. the computer will then write the data to the destination disk. Then the computer will instruct you to place the source disk back in the disk drive:

```
INSERT SOURCE DISKETTE AND PRESS ENTER
```

Remove the destination disk from the disk drive, insert the source disk, close the drive door, and press the "ENTER" key. The computer will then read some more data from the source disk.

This procedure will continue until all of the sectors of the source disk have been read into memory and written to the destination disk. Be careful not to get the two disks mixed up during this procedure!

SYNTAX :

```
CLOSE [[#] buffer][,[#] buffer],...
```

PURPOSE :

The CLOSE statement is used to terminate disk I/O operations. All disk I/O is handled through buffers (reserved areas of memory) assigned as needed with the OPEN command.

When a disk read statement (INPUT or GET) is executed, the required disk sector is read into the associated memory buffer. Subsequent INPUT statements will access data in the buffer until the requested data lies in another sector. Then a new sector may be read into the buffer. This technique increases the throughput of disk operations by minimizing disk accesses.

Disk write commands are handled in much the same way. Data to be written to the disk are buffered in memory until the buffer is full, or until another is requested. At this point, the entire buffer will be written to the appropriate sector on the disk, thus minimizing disk I/O operations.

When an input file buffer is closed, the only action taken by the computer is to free the affected buffer space for subsequent use with another file. When an output file buffer is closed, the computer will write the buffer contents (if any) to the disk file, and update the directory entry to reflect the new file size. The output buffer space is then available for use with another file.

If a buffer number is specified with the CLOSE command, only that buffer will be closed. If no buffer is specified, all currently open buffers will be closed.

EXAMPLE :

```
CLOSE 1, 3

CLOSE #1, #2

CLOSE
```

In the first example, only buffers 1 and 3 will be closed. In the second example, only buffers 1 and 2 will be closed. In the final example, all currently open buffers will be closed.

SYNTAX :

```
     COPY <"filename1/ext[:d]" TO <"filename2/ext[:d]">
```

or

```
     COPY <"filename/ext[:d]">
```

PURPOSE :

The copy command is used to make a duplicate copy of a file. The contents of the source file "filename1" are copied to the destination file "filename2". If the drive parameter [:d] is not specified, the current default drive will be used.

After the computer has completed execution of the COPY command, the contents of the destination file will be identical to the contents of the source file, although the file names may be different.

EXAMPLE :

```
     COPY "MYPROG/BAS" TO "YOURPROG/BAS"

     COPY "MYPROG/BAS:Ø" TO "MYPROG/BAS:1"
```

In the first example, the contents of the file "MYPROG/BAS" are copied to the file "YOURPROG/BAS". The source file must be on the default drive, and the destination file will be written on the default drive. In the second example, the file "MYPROG/BAS" on drive Ø will be copied to the file "MYPROG/BAS" on drive 1.

WARNING: In some cases, the COPY command may destroy a program in memory. It is good practice to not use the COPY command when there is a program in memory.

CAUTION: Be careful not to copy a file to a destination with the same name on the same disk. This will potentially result in the file being deleted!

USING THE COPY COMMAND ON SINGLE-DRIVE SYSTEMS

EXAMPLE :

```
     COPY "MYPROG/BAS"
```

This is the correct syntax for copying files from one disk to another on single-drive systems. When the command is entered, the computer first reads a number of sectors from the source disk and then prompts with :

```
     INSERT DESTINATION DISK AND PRESS ENTER
```

Remove the source disk, insert the disk on which you want the copy written, close the drive door, and press the "ENTER" key. The computer will write the file portion in memory to the destination disk. Depending on the length of the file, the computer may prompt you with:

```
     INSERT SOURCE DISK AND PRESS ENTER
```

If this occurs, remove the destination disk and replace the source disk in the drive. The computer will repeat this procedure until all of the file has been copied to the destination disk.

SYNTAX :

```
CVN <(data)>
```

PURPOSE :

The CVN function will decode a number which has been encoded into a string with the MKN$ function (refer to the description of the MKN$ function). The CVN function will decode a 5-byte string code created by the MKN$ function back into a number.

The CVN function is normally used in conjunction with direct access file input since all numeric data stored in a direct access file must be converted to string data using the MKN$ function. All numeric data read from a direct access file must be converted to numeric form using the CVN function before it can be operated on as numeric data.

EXAMPLE :

```
A= CVN (A$)
```

In this example, the variable "A" is assigned the numeric equivalent of the 5-byte encoded string A$. The encoded data in A$ was created by the MKN$ function, or read from a direct access file.

SYNTAX :

```
DIR [drive]
```

PURPOSE :

The DIR command is used to display the directory of a disk. The directory of the disk in [drive] will be displayed. If the [drive] parameter is omitted, the directory of the disk in the current default drive will be displayed.

EXAMPLE :

```
DIRØ

     MYPROG  BAS   Ø B 3
     FILE    DAT   1 A 4
     TEST    BIN   2 B 1
```

The directory listing consists of a 5 column display. The information presented in each column is explained below:

Column	Meaning	
1	Filename	
2	File Extension	
3	File Type :	Ø => BASIC program file 1 => BASIC data file 2 => MACHINE LANGUAGE file 3 => EDITOR source file

4	Storage Format :	A => ASCII
		B => Binary

5	File Length (in granules)	

SYNTAX :

```
DRIVE <drive>
```

PURPOSE :

The DRIVE command is used to change the default drive. When the computer is RESET, the default drive will be Ø. Drive Ø will remain the default drive until, and unless, you change the default drive with the DRIVE command. The drive number can only be changed to Ø, 1, 2, or 3.

The default drive will be used for all commands where a drive number is required, except where a drive number is specified on the command line. Thus, the DRIVE command is a convenience feature. If you are doing a lot of work on drive 1, for example, you may use the command "DRIVE 1" to change the default drive to drive 1. Then, you need not use the ":1" parameter on each command line.

The default drive will remain in effect until you change it, or until the computer is RESET.

EXAMPLE :

```
DRIVE 1
```

In this example, the default drive is changed to drive 1. Drive 1 will remain the default drive until the DRIVE command is used to change it, or until the computer is RESET.

SYNTAX :

```
     DSKINI <drive>[,n]
```

PURPOSE :

The DSKINI command is used to format a disk. A new disk must be formatted before it can be used by the computer. A used disk can be formatted to erase all of the data on the disk.

The DSKINI command will format the disk into 35 tracks each with 18 granules.

The optional part of the command is the skip-factor which is normally left out (4 is used in this case). The skip-factor is set up to allow the computer time to process the data between reading sectors. The skip-factor can be changed for certain special purposes.

EXAMPLE :

```
     DSKINI Ø
```

SYNTAX :

```
DSKI$ <drive>,<track>,<sector>,<var1$>,<var2$>
```

PURPOSE :

The DSKI$ statement is a special disk I/O statement which allows data to be input directly from the disk. The DSKI$ statement will input one sector (256 bytes) directly from the specified disk. The data will be input into two string variables, <var1$> and <var2$>. The string variable <var1$> will contain the first 128 bytes of the sector, while the string variable <var2$> will contain the second 128 bytes of the sector.

The <drive>, <track>, and <sector> parameters are required by the DSKI$ statement. The <drive> parameter specifies the drive to be read (Ø-3). The <track> parameter specifies the track to be read (Ø-34). The <sector> parameter specifies the sector to be read (1-18).

EXAMPLE :

```
DSKI$ Ø,17,3,A1$,A2$
```

In this example, the data from track 17, sector 3, of the disk in drive Ø will be read into variables A1$ and A2$. The first 128 bytes of the sector will be read into A1$, and the remaining 128 bytes will be read into A2$.

SYNTAX :

```
DSKO$ <drive>,<track>,<sector>,<"data1">,<"data2">
```

PURPOSE :

The DSKO$ statement is a special disk I/O statement which allows data to be written directly to the disk. The DSKO$ statement will put up to 256 bytes directly to a specified sector. The first string <"data1"> will be written in the first half of the sector (bytes Ø-127), while the second string <"data2"> will be written on the second half of the sector (bytes 128-255). The data represented by <"data1"> and <"data2"> may be either string data or variables.

The DSKO$ statement requires the parameters <drive>, <track>, and <sector>. The <drive> parameter specifies on which drive (Ø-3) the data is to be written. The <track> parameter specifies the track (Ø-34) on which the data is to be written. The <sector> parameter specifies the sector (1-18) on which the data will be written.

CAUTION: This statement writes data directly to the disk, bypassing the file control system. Be careful not to destroy data on the disk!

EXAMPLE 1:

```
DSKO$ Ø,15,1Ø,"string data 1","string data 2"
```

EXAMPLE 2:

```
A$ = "string data 1"
B$ = "string data 2"
DSKO$ Ø,15,1Ø,A$,B$
```

In both examples, the string "string data 1" will be written on the first half of sector 1Ø on track 15 of the disk in drive Ø. The string "string data 2" will be written on the second half of the sector.

SYNTAX :

```
     EOF ((buffer))
```

PURPOSE :

The EOF function returns the value -1 if there is no more data to be read from a file buffer. If there is more data, the value Ø is returned. So, the value returned by the EOF function can be tested to determine if all of a file has been read.

EXAMPLE :

```
     1Ø OPEN "I", #1, "TEST/DAT"

     2Ø I = I + 1

     3Ø IF EOF(1) = -1 THEN 8Ø

     4Ø GET #1, I

     5Ø INPUT #1, A$

     6Ø PRINT A$

     7Ø GOTO 2Ø

     8Ø CLOSE #1
```

This program segment will read all of the data from the file "TEST/DAT". Each record read is printed until the end-of-file tag is encountered. At that point, the file will be closed.

SYNTAX :

```
FIELD <#buffer>,<field size> AS <field name>,...
```

PURPOSE :

The FIELD statement is used in conjunction with direct-access (sometimes called "random access") files. When the field statement is used, it must be executed before the GET or PUT statements (see also GET and PUT). The FIELD statement allocates space for variables in the direct-access file buffer.

The <#buffer> parameter specifies the buffer number to which the field statement applies. This parameter is specified only once per FIELD statement. The <field size> parameter specifies the number of bytes to be reserved for the variable <field name>. Buffer space will be allocated in the order that the fields are identified in the FIELD statement.

EXAMPLE :

```
FIELD #1, 5 AS A1$, 10 AS A2$, 7 AS B$
```

In this example, the first 5 bytes of disk I/O buffer #1 will be reserved for variable "A1$". The next 10 bytes will be reserved for the variable "A2$", and the next 7 bytes will be reserved for the variable "B$". Note that the FIELD statement does not place data in the random file buffer (refer to the LSET, RSET, and PUT statements).

Any number of FIELD statements may be executed for a given file in order to assign buffer space. That is, if there is more space to be assigned than will fit in one FIELD statement, you may use multiple FIELD statements to assign all of the variables in a given file buffer.

SYNTAX :

```
FILES <number>[,size]
```

PURPOSE :

When the computer is turned on or RESET, memory space is automatically reserved for two file buffers (#1 and #2, 256 bytes each), and 256 bytes are reserved for the direct-access record buffer. The file buffers are used for all types of file access (sequential or direct-access, input or output). The direct-access record buffer is used to hold direct-access records during file I/O.

In direct-access file I/O, data are read one sector at a time into the associated file buffer. Then, each record is moved from the file buffer into the direct-access record buffer.

The FILES statement may be used to specify the amount of memory space to be reserved for file I/O buffers as well as for the direct-access record buffer. The <number> parameter specifies the numbers of buffers (1-15) to be reserved. A total of 256 bytes will be reserved for each file buffer. The [size] parameter is optional, and, if included, specifies the amount of memory to be reserved for the direct-access record buffer. If the [size] parameter is not included, the computer will reserve 256 bytes of memory for direct buffer space, regardless of how many buffers there may be.

If you plan to have more than 2 files open at one time, then you must use the FILES statement to reserve the appropriate amount of buffer space. Likewise, if you plan to use one or more direct-access files, and the combined record length is in excess of 256 bytes, you must use the FILES statement to allocate the appropriate amount of buffer space.

EXAMPLE :

```
FILES 3
FILES 5,1000
```

In the first example, buffer space will be reserved for 3 files, and 256 bytes will be reserved for the direct-access record buffer. In the second example, buffer space will be reserved for 5 files, and 1000 bytes will be reserved for the direct-access record buffer.

FREE FUNCTION

SYNTAX :

```
     FREE<(drive)>
```

PURPOSE :

The FREE function returns the amount of free (unused) disk space remaining on the disk in the specified drive. The <drive> parameter is required, and specifies which drive the computer is to check (Ø-3).

The FREE function will return an integer value representing the number of unallocated granules on the disk (each granule contains 9 sectors of 256 bytes of data). There are 68 granules available on an unused disk. This is equal to 156672 bytes per disk.

EXAMPLE :

```
     PRINT FREE (Ø)
```

In this example, the computer will print the number of free (unallocated) granules remaining on the disk in drive Ø.

SYNTAX :

```
GET <#buffer>[,record]
```

PURPOSE :

The GET statement is used to read one record from a direct-access (random access) disk file to the associated input buffer. Note that the OPEN statement must first be executed to associate an input buffer with a disk file.

The <#buffer> parameter is required, and specifies which buffer (1-15) is to receive the data. The [record] parameter is optional, and, if included, specifies which file record is to be read. If the [record] parameter is not specified, the next sequential record will be read. That is, the record number of the record currently in the buffer will be incremented by one, and the record corresponding to the new record number will be read into the buffer.

EXAMPLE :

```
GET #1, 2

GET #3, N
```

In the first example, the computer will read record number 2 from the previously OPEN'd file into buffer #1. In the second example, record number "N" of the previously OPEN'd file is read into buffer #3. As shown in the second example, the record number may be specified through an integer variable. In this way, the record number to be read may be assigned under program control.

SYNTAX :

```
INPUT <#buffer>,<var1>[,var2][,var3],...
```

PURPOSE :

The INPUT statement is used to move data from a disk input buffer to a program variable. Data items in the buffer are assigned to the variable names in the order that they are encountered on the INPUT statement line. Variable <var1> will be assigned the first data item in the buffer, variable [var2] will be assigned the second item, and so on.

The INPUT statement requires a minimum of two parameters. The <#buffer> parameter specifies the buffer from which the data are to be obtained (1-15). The buffer number must be previously referenced in an OPEN statement. The variable parameters (<var1>, [var2],...) specify the variable names into which the buffer data are moved. At least one variable name must be included on the INPUT statement line.

The INPUT statement requires a minimum of one variable name. As many variable names as required may be named on the command line, however the total line length may not exceed 255 characters. The data from the buffer will be assigned to the variable names in the order that they appear on the command line.

Note that in the case of direct-access file input operations, the GET statement must be executed prior to the INPUT statement. Otherwise, there is no valid data in the direct-access record buffer, and the results of the INPUT statement will be meaningless.

EXAMPLE :

```
INPUT #1, A$, B$, C$
```

In this example, the data in buffer #1, which has been previously OPEN'd, will be assigned to the variables A$, B$, and C$. The first data item in the buffer will be assigned to the variable "A$", the second item will be assigned to the variable "B$", and the third datum will be assigned to the variable "C$".

SYNTAX :

```
     KILL <"filename/ext[:d]">
```

PURPOSE :

The KILL command is used to delete a file from a disk. The file name and extension must be specified. If the drive parameter [:d] is specified, the file named will be deleted from the disk in the specified drive. If the drive specification is omitted, the file named will be deleted from the disk in the default drive.

Note that the filename, the extension, and the drive (if specified) must all be enclosed in quote marks. When the file is deleted, the disk space allocated to the file will be returned to the free-space pool providing more usable disk space.

EXAMPLE :

```
     KILL "MYPROG/BAS"

     KILL "MYPROG/BAS:1"
```

In the first example, the file named "MYPROG/BAS" will be deleted from the disk in the current default drive. In the second example, the file named "MYPROG/BAS" will be deleted from the disk in drive 1.

SYNTAX :

```
LINE INPUT <#buffer>,<vari$>
```

PURPOSE :

The LINE INPUT statement is used to read a "line" of data from a disk I/O buffer. The buffer must be assigned to a file by a previously executed OPEN statement, and there must be data in the buffer. A "line" of data is defined as a group of data terminated by a "ENTER" character (ØD hex, 13 decimal). All data up to, but not including, the "ENTER" character will be transferred to the variable named on the statement line.

Two parameters are required in conjunction with the LINE INPUT statement. The <#buffer> parameter specifies the input buffer (1-15) from which the data are to be taken. The <vari$> parameter specifies the string variable name into which the data are to be transferred.

EXAMPLE :

```
LINE INPUT #1, B$
```

In this example, the data from input buffer #1 will be transferred to the string variable "B$". All data up to the first "ENTER" character will be transferred from the buffer to the string variable.

SYNTAX :

```
     LOAD <"filename[/ext][:d]">[,R]
```

PURPOSE :

The LOAD command is used to transfer a BASIC program file from disk to main memory. When a program is loaded into memory, the program and data (if any) currently in main memory will be written over by the new program. Be careful not to load a file over a program that you want to keep!

The LOAD command requires only one parameter: the file name. The file name extension is optional. If the file name extension is not included, the extension "/BAS" will be used.

The drive parameter [:d] is also optional. If included, the file will be loaded from the disk in the drive specified. Otherwise, the current default drive will be used.

The "R" parameter is an optional parameter. If included, the "R" parameter will instruct the computer to run the program immediately after it is loaded. This eliminates the need to enter the "RUN" command after the file is loaded.

EXAMPLE :

```
     LOAD "MYPROG"

     LOAD "MYPROG/BAS:2",R
```

In the first example, the file named "MYPROG/BAS" on the disk in the current default drive will be loaded into memory. In the second example, the file "MYPROG/BAS" on the disk in drive 2 will be loaded into main memory and will run immediately after it is loaded.

SYNTAX :

```
     LOADM <"filename[/ext][:d]">[,offset]
```

PURPOSE :

The LOADM command is used to transfer a machine code program from disk to main memory. When the LOADM command is used, the program and/or data (if any) at the load address will be written over by the new program. Be careful not to destroy any program or data that you wish to keep!

The LOADM command requires only one parameter: the file name. The file name extension is optional. If the file name extension is not included, the computer will assume the extension of "/BIN".

The drive parameter [:d] is also optional. If included, the file will be loaded from the disk in the drive specified. Otherwise, the current default drive will be used.

The [offset] parameter is an optional parameter. If included, the offset specified will be added to the program load address. The program will then be loaded at the address determined by the sum of the offset and the load address. Note that the offset parameter is assumed to be in decimal unless preceded by the characters "&H" to indicate that the number is hexadecimal.

EXAMPLE :

```
     LOADM "MYPROG"

     LOADM "MYPROG/BIN:1",&H1ØØØ
```

In the first example, the file named "MYPROG/BIN" on the disk in the current default drive will be loaded into memory. The program will be loaded at the program's normal load address (as specified in the file). In the second example, the program named "MYPROG/BIN" will be loaded from the disk in drive 1. The program will be loaded at the address determined by the sum of the program's normal load address and 1ØØØ Hex.

SYNTAX :

```
     LOC ((buffer))
```

PURPOSE :

The LOC function returns the record number of the file record currently in the specified buffer. It is assumed that a buffer has been previously opened (see OPEN statement), and that a record has been placed in the buffer.

Only one parameter is required in conjunction with the LOC function: the (buffer) parameter. This parameter specifies the buffer number (1-15) for which the record number is to be returned.

EXAMPLE :

```
     PRINT LOC(1)

     A = LOC(1)
```

In the first example, the record number of the file record currently in buffer #1 will be printed. In the second example, the record number of the file record currently in buffer #1 will be assigned to the variable A.

SYNTAX :

LOF <(buffer)>

PURPOSE :

The LOF function returns the last record number of a direct-access file. The specified buffer must be associated with a direct-access file name previously executed OPEN statement. This function is especially useful when reading a file of unknown length.

The LOF function requires only one parameter: the <buffer> parameter. This specifies the buffer number (1-15) of the buffer associated with the file for which the last record number is to be returned. Note that the file name is associated with a buffer only in the OPEN statement, so the OPEN statement must have been previously executed.

EXAMPLE :

```
10 OPEN "D", #1, "EXAMPLE/TXT"

20 FOR R = 1 TO LOF(1)

30 GET #1, R

40 INPUT #1, E1$

50 PRINT E1$

60 NEXT R

70 CLOSE #1
```

In this example, the entire file will be read and printed. The FOR loop in line 20 will increment R, from 1 to the last record of the file.

SYNTAX :

```
LSET <field> = <data>
```

PURPOSE :

The LSET statement assigns data to the field name specified, and left-justifies the data. The field name must have been previously defined in a FIELD statement. The data will be moved from the variable or direct assignment made in the LSET statement <data> to the memory area reserved for the variable specified by the <field> parameter.

The LSET statement will left-justify the data as it is transferred to the field variable. If the data transferred is too long for the field variable as defined in the FIELD statement, the data will be truncated.

The LSET statement may be used only with string data. If numeric data is to be LSET, it must be converted to string data first. (use MKN$)

EXAMPLE :

```
10 FIELD #1, 6 AS A1$, 10 AS A2$

20 LSET A1$ = "STRING 1"

30 LSET A2$ = B$
```

In this example, the field variable A1$ is assigned a length of 6 characters, and A2$ is assigned a length of 10 characters (both are defined by the FIELD statement in line 10). In line 20, the data "STRING 1" is transferred and left-justified into variable A1$. However, since the variable A1$ was assigned a length of 6 characters, and the data is 8 characters long, the data will be truncated. The result is that the variable A1$ will contain the data "STRING".

In line 30, the data contained in the variable "B$" will be transferred to, and left-justified in, the variable A2$. As in line 20, if the data contained in the variable "B$" is longer than 10 characters (the assigned length of A2$), then A2$ will contain only the first 10 characters of the data from B$.

SYNTAX :

```
     MERGE <"filename[/ext][:d]">[,R]
```

PURPOSE :

The MERGE command is used to transfer a basic program file from disk to memory, merging it with the program already in memory. Only a program which was saved with the ASCII ("A") option may be merged (refer to the SAVE command).

When two programs are merged, the result is that the program in memory will contain all of the lines from both programs. The only exception is that when there are line number conflicts (the same line number exists in both programs), the line from the disk file will replace the line already in memory.

The parameter "filename" is required. The file name extension "/ext" is optional. If the file name extension is not specified, the extension "/BAS" will be used.

The drive parameter [:d] is also optional. If the drive parameter is not specified, the current default drive will be used.

The "R" parameter is optional. If included on the command line, the "R" parameter will cause the program to be run immediately after the merge is complete.

EXAMPLE :

```
     MERGE "MYPROG"

     MERGE "MYPROG/BAS:1",R
```

In the first example, the program file "MYPROG/BAS" will be merged with the program currently in memory. The program file must be on the disk in the current default drive. In the second example, the program file named "MYPROG/BAS" on drive 1 will be merged with the program currently in memory. The resulting program will be run immediately after the merge is complete.

SYNTAX :

```
MKN$ <(data)>
```

PURPOSE :

The MKN$ function will convert a numeric variable, or a numeric constant, to a 5-byte coded string. This function is opposite to the CVN function. The CVN function may be used to decode the string back to a number.

The MKN$ function is especially useful in direct access file operations, since all direct access file output must be in the form of string variables.

EXAMPLE :

```
A$ = MKN$ (123)

B$ = MKN$ (C)
```

In the first example, the constant 123 will be converted to a 5-byte string code and stored in the variable "A$". In the second example, the number stored in the variable "C" will be converted to a 5-byte string code and stored in the variable "B$".

SYNTAX :

```
OPEN <"mode">,<#buffer>,<"filename[/ext][:d]">[,record length]
```

PURPOSE :

All disk file I/O is handled through buffers (reserved area in memory). The OPEN statement associates a specific disk file with a specific I/O buffer. The OPEN statement must be executed prior to any file I/O (GET, PUT, INPUT#, PRINT#, etc).

The "mode" parameter specifies the mode of operation for the file buffer:

"I" - Input from sequential access file
"O" - Output to sequential access file
"D" - Input and/or output to/from direct access file

The <#buffer> parameter specifies the buffer number to be associated with the file named on the statement. The <#buffer> parameter may be any number in the range 1 to 15, but must not be the number of a buffer already open. Also if buffer numbers other than #1 or #2 are to be used, the FILES statement must be executed prior to the OPEN statement.

The "filename" specifies the file name to be associated with <#buffer> in all subsequent file I/O operations until the file is closed with the CLOSE statement. If "/ext" is omitted, the extension "/DAT" will be used. If the drive parameter [:d] is omitted, the current default drive will be used.

The [record length] parameter is optional, and applies only to direct access files (mode "D"). If the [record length] is not specified, the record length will default to 256 bytes. This parameter may not be used with sequential access files.

EXAMPLE :

```
OPEN "I", #1, "TEST"

OPEN "D", #", "DATA/TST:1", 40
```

In the first example, the file "TEST/DAT" on the disk in the default current drive is opened for input through buffer #1. In the second example, the file "DATA/TST" on the disk in drive 1 is opened for direct access I/O through buffer #2. The record length for this file is specified as 40 bytes.

SYNTAX :

```
     PRINT <#buffer>,<data1>[,data2],...
```

PURPOSE :

The PRINT statement moves the data named on the statement line to the disk output buffer named on the statement line. The data are formatted in the same way as when a normal PRINT statement is executed. The only difference is that the data are sent to the disk file buffer rather than to the screen or the printer. Refer to the description of the PRINT statement in the Radio Shack book "Going Ahead with Extended COLOR BASIC" for details.

Note that the data items named on the statement line may be separated by either commas (",") or semicolons (";"). As with the normal PRINT statement, the comma will perform a tab function by inserting spaces between data items. The semicolon will supress the spaces (and the ENTER character) between data.

EXAMPLE :

```
     PRINT #1, "THIS IS A TEST", S

     PRINT #5, A1$; A2$; N; L
```

In the first example, the string "THIS IS A TEST" and the contents of the variable "S" will be output to the disk file associated with buffer #1. Because of the comma, a number of spaces will be output between the string and the contents of the variable "S". If a semicolon had been used instead of a comma, the spaces between the string and the data from the variable "S" would have been supressed.

In the second example, the contents of the variables "A1$", "A2$", "N", and "L" will all be output to the disk file associated with buffer #5. Since the variables are delimited by semicolons, no spaces will be output between the variables.

SYNTAX :

```
PRINT <#buffer>, USING <"format">;<data1>[,data2],...
```

PURPOSE :

The PRINT USING statement moves the data named on the statement line to the disk output buffer named on the statement line. The data are formatted in the same way as when a normal PRINT USING statement is executed. The only difference is that the data is sent to the disk file instead of the screen or printer. Refer to the description in Radio Shack's book "Going Ahead with Extended COLOR BASIC" for more details.

The <format> parameter specifies the data format to be used when the data are output, in the same way as data are formatted when sent to the screen or printer. The symbols used for formatting are summarized below.

#	Specifies a numeric field.
.	Places a decimal point position.
,	Places a comma between each 3 digits in a numeric field.
**	Fills leading spaces of a field with asterisks.
$	Places a dollar sign at the beginning of a field.
$$	Places "floating" dollar sign adjacent to the first digit in a numeric field.
+	If placed in the first position of a numeric field, it indicates sign to be printed in front of the number. If placed in the last position of a numeric field, it indicates the sign is to be printed after the number.
^	Indicates number to be printed in exponential format.
-	Places minus sign after negative numbers.
%	Delimits literal (string) fields.
!	Indicates use of only the first character of a string.

EXAMPLE :

```
PRINT #1, USING "###.##"; 197.664
```

This example would output the data "197.664" as "197.66". The third digit after the decimal point will be truncated since the specified format indicates only two decimal positions.

EXAMPLE :

```
PRINT #3, USING "**$###.##-"; -3.78
```

This example would result in the output of "**$3.78-" to the file associated with buffer #3.

EXAMPLE :

```
PRINT #1, USING "% %"; "STRING"
```

This example would output the data "STR" to the disk file associated with buffer #1.

SYNTAX :

```
PUT <[#]buffer>[,record number]
```

PURPOSE :

The PUT statement is used to write a direct access file buffer to the associated disk file. When performing direct access file output, the WRITE statement transfers data to the disk buffer, and the PUT statement transfers the buffer data to the disk file record.

The <buffer> parameter is required on the PUT statement line, but the [record number] parameter is optional (note that the symbol "#" is optional in the <buffer> parameter). The [record number] parameter specifies the record number of the file to which the buffer is to be written. If this parameter is omitted, the record number which was last read into the buffer will be used.

EXAMPLE :

```
10 OPEN "D", #3, "DATA/DAT", 10

20 FOR R = 1 TO 20

30 WRITE #1, "          "

40 PUT #3, R

50 NEXT R

60 CLOSE #3
```

This program segment will fill 10 records of the file "DATA/DAT" with ASCII spaces. Note that the file consists of 20 records of 10 bytes each.

SYNTAX :

```
RENAME <"filename1/ext[:d]"> TO <"filename2/ext[:d]">
```

PURPOSE :

The RENAME command is used to give an existing file a new name. The file contents will not be moved or changed in any way. Only the name of the file in the directory will be changed.

The "filename1" parameter specifies the current file name. The "filename2" parameter specifies the new file name. So, the RENAME command will change the current file name "filename1" to the new file name "filename2".

The file name extension "/ext" is required for correct use of this command.

The drive parameter [:d] is optional. If it is included, The file will be renamed on the disk in the drive specified. Otherwise, the file named is assumed to be on the disk in the current default drive.

EXAMPLE :

```
RENAME "MYPROG/BAS" TO "YOURPROG/BAS"
```

In this example, the file named "MYPROG/BAS" on the disk in the current default drive is renamed to "YOURPROG/BAS".

SYNTAX :

RSET <field> = <data>

PURPOSE :

The RSET statement assigns to the field name <field> and right-justifies the data. The field name <field> must have been previously defined in a FIELD statement. The data will be moved from the variable or direct assignment <data> on the statement line to the memory area reserved for the variable <field>.

The RSET statement will right-justify the data as it is transferred to the field variable. If the data transferred are too long for the field variable as defined in the FIELD statement, the data will be truncated.

The RSET statement may be used only with string data. If numeric data is to be RSET, it must first be converted to string data (refer to the MKN$ function).

EXAMPLE :

```
10 FIELD #1, 10 AS A1$, 5 AS A2$

20 RSET A1$ = "STRING 1"

30 B$ = "STRING 2"

40 RSET A2$ = B$
```

In this example, the variable "A1$" has been fielded as a 10 character variable, and "A2$" has been fielded as a 5 character variable (line 10). In line 20, variable "A1$" is RSET to "STRING 1". The result would be that A1$ = " STRING 1". Two spaces would be padded to the left of the data "STRING 1" so that the string is right-justified within the 10 character field.

In line 40, A2$ is RSET to the variable B$ which contains the string "STRING 2". Since "STRING 2" consists of 8 characters, and the variable A2$ has been fielded as a 5 character variable (line 10), the contents of the field variable A2$ after execution of line 40 would be "STRIN".

SYNTAX :

```
     RUN <"filename[/ext][:d]">[,R]
```

PURPOSE :

The RUN command is used to load and run a BASIC program file from disk. The BASIC program file "filename" will be loaded from disk into memory and executed. If the extension parameter "/ext" is omitted, the extension "/BAS" is assumed. If the drive parameter [:d] is omitted, the program file is assumed to be on the disk in the current default drive.

If the ",R" option is used, all currently open disk files will remain open.

EXAMPLE :

```
     RUN "MYPROG"

     RUN "MYPROG/TST:1"
```

In the first example, The program saved in the file named "MYPROG/BAS" on the disk in the current default drive will be loaded and executed. In the second example, the program saved in the file named "MYPROG/TST" on the disk in drive 1 will be loaded and executed.

SYNTAX :

```
     SAVE <"filename[/ext]:d]">[,A]
```

PURPOSE :

The SAVE command is used to save a basic program to disk. The BASIC program currently in memory will be saved on a disk file with the name "filename" as the specified on the command line. If the extension "/ext" is not included on the command line, the extension "/BAS" will be used. If the drive parameter [:d] is omitted from the command line, the program will be saved on the disk in the current default drive.

If the ",A" parameter is included on the command line, the program will be saved in ASCII form (not TOKENISED) on the disk. If the ",A" parameter is omitted from the command line, the program will be saved in a coded (TOKENISED) form. Although a program saved in ASCII will require more disk space, it may be edited (using a word processor) like any other ASCII text file. Also, a program which is to be merged with another file must be saved in ASCII form (refer to the description of the MERGE command).

EXAMPLE :

```
     SAVE "MYPROG"

     SAVE "MYPROG/TST:1",A
```

In the first example, The program currently in memory will be saved in a file named "MYPROG/BAS" on the disk in the current default drive. In the second example, the program currently in memory will be saved in ASCII format in a file named "MYPROG/TST" on the disk in drive 1.

SYNTAX :

```
SAVEM <"filename[/ext][:d]">,<first>,<last>,<execution>
```

PURPOSE :

The SAVEM command is used to save a machine-code program or data to disk. The program or data currently in memory will be saved in a file named "filename". If the file name extension "/ext" is not specified, the extension "/BIN" will be used. If the drive parameter [:d] is not specified, The program will be saved on the disk in the current default drive.

The <first> parameter specifies the first (lowest) address of the program to be saved. The <last> parameter specifies the last (highest) address of the program. The <execution> parameter specifies the entry point (execution address) of the program. Note that the <first>,<last>, and <execution> address parameters are assumed to be in decimal unless they are preceded by "&H" which indicates that the numbers are HEXADECIMAL.

EXAMPLE :

```
SAVEM "MYPROG",1024,2048,1024

SAVEM "MYPROG/TST:1", &H1000, &H1400, &H1020
```

In the first example, the program located in memory between addresses 1024 decimal and 2048 decimal will be saved in a file named "MYPROG/BIN" on the disk in the current default drive. The execution address (entry point) of this program is defined to be 1024 decimal. In the second example, the program in memory between 1000 HEX and 1400 HEX will be saved in a file named "MYPROG/TST" on the disk in drive 1. The execution address of this program is specified as 1020 HEX.

SYNTAX :

```
UNLOAD [drive]
```

PURPOSE :

The UNLOAD command is used to close all files which are currently open on the specified drive. If you change disks while one or more files are open, it is very likely that you will destroy the directory on the disk. For this reason, it is good practice to use the UNLOAD command before changing disks. Of course, if you are not doing any file I/O, you don't need to use the UNLOAD command.

The [drive] parameter on the command line is optional. If the drive parameter is included, the files which are open on the drive specified will be closed. If the drive parameter is omitted from the command line, the files which are open on the current default drive will be closed.

EXAMPLE :

```
UNLOAD

UNLOAD 1
```

In the first example, all of the files open on the current default drive will be closed. In the second example, all of the files open on drive 1 will be closed.

SYNTAX :

```
     VERIFY ON
```

or

```
     VERIFY OFF
```

PURPOSE :

The VERIFY command is used to control the disk I/O read-after-write verify function. Normally, the computer does not verify disk write operations. When the command "VERIFY ON" is invoked, the computer will perform read-after-write verification of all data written to disk. This verification function will remain in effect until either the command "VERIFY OFF" is executed, or the computer is RESET.

When the computer performs a read-after-write verification, it will read a sector back just after it is written to the disk. It will then compare what it read from the sector with what it wrote to the sector. If the two are not the same, the computer will attempt to write the sector again, up to a total of five times. If it still cannot record the data properly it will issue a VF ERROR message. In this way, you may be sure that the data was written to disk properly, before it is too late.

EXAMPLE :

```
     VERIFY ON
```

In this example, the read-after-write verification function is enabled. It is good practice to use this feature at all times.

SYNTAX :

```
     WRITE <#buffer>,<data1>[,data2],...
```

PURPOSE :

The write statement is used to transfer data from constants or variables to a specified disk output buffer. The data items will be transferred to the buffer in the order that they appear on the statement line. If the specified buffer is associated with a direct-access file, the data items will be placed in the buffer in fields immediately adjacent to one another. Data items in a sequential-access file will be separated by commas.

The <#buffer> parameter specifies the buffer (1-15) to which the data are to be transferred. At least one data item must be present on the statement line. The data item may be either a constant or a variable. If more than one data item is specified, the items must be separated by commas. Unlike the PRINT statement, the commas on the write statement line perform no function other than delimiting the items on the list.

EXAMPLE :

```
     WRITE #1, A, B, B$, "STRING"
```

In this example, the contents of the variables "A", "B", and "B$", as well as the string constant "STRING", will be transferred to disk output buffer #1. The contents of the variable "A" will be placed in the buffer first, followed by the contents of "B", the contents of "B$", and finally by the constant "STRING".

DISK EXTENDED BASIC COMMANDS

A summary.

Here are the instructions used in DISK BASIC and a brief explanation of each one.

```
BACKUP <source drive> TO <destination drive>
BACKUP Ø
```

Copies the entire contents of disk in <source drive> to disk in <destination drive>. The first command listed is used on multiple drive systems or on double sided drives (not supplied by TANDY), to transfer files from one drive to another or one side to another, e.g. BACKUP Ø to 2. The second command is for single disk drive users. In this mode the computer will prompt you to change disks when necessary.
DO NOT MIX UP THE DISKS !!!

```
CLOSE [[# buffer][,[#] buffer],...
```

Closes disk files associated with the buffers specified. If no buffer specified, all buffers are closed.

```
COPY <"filename 1/ext [:d]"> TO <"filename 2/ext [:d]">
COPY <"filename/ext">
```

Contents of source file "filename 1" are copied to destination file "filename 2". Second form is for single disk drive users to copy a file to another disk. COPY can be used to create an identical file on the same disk as long as the filenames are different.

```
CVN <(data)>
```

Decodes a 5 character string variable to numeric form. E.g. A=CVN(A$). See also MKN$

```
DIR [drive]
```

Displays a directory of the disk in the specified drive. If no drive is specified then the default drive is used. Default drive is Ø on power up. E.g. DIR 2

```
DRIVE <drive>
```

Changes the default drive number to <drive>. Default drive is used in an instruction where no drive is specified. E.g DRIVE 2.

```
DSKINI <drive>
```

This command is used to format a new disk or clean off a used disk.
WARNING : USE WITH CARE.
THIS COMMAND ERASES EVERYTHING FROM YOUR DISK !!!!!

```
DSKI$ <drive>,<track>,<sector>,<string variable 1>,<string variable 2>
```

Inputs <sector> of <track> on disk <drive> directly into <string variable 1> and <string variable 2>. String variable 1 will receive the first 128 bytes of the sector and string variable 2 will receive the second 128 bytes.

DSKO$ <drive>,<track>,<sector>,<"data 1">,<"data 2">

Writes <"data 1"> and <"data 2"> directly into <sector> of <track> on disk in <drive>. <"data 1"> or <"data 2"> can be string variables. The first data or string variable is written to the first half of the sector and the second data or variable is written to the second half.

EOF <(buffer)>

Returns the value of -1 if there is no more data to be read from the disk file. Returns Ø if there is more data. E.g. IF EOF(1) = -1 THEN 6Ø

FIELD <#buffer>,<field size> AS <fieldname>,...

Associates variable names with, and defines fields in disk file buffer. Parameter <#buffer> specifies buffer number (1-15), <field size> specifies the number of characters to be reserved for variable <fieldname>. Fields are reserved in the order they are specified. E.g. FIELD#1,5 AS A$,1Ø AS B$,3 AS C$

FILES <number>,[size]

Reserves memory for <number> of file buffers, and [size] in characters of each. Used for DIRECT ACCESS record buffers only. If [size] is not specified, 256 characters will be reserved. System defaults to FILES 2,256.

FREE <(drive)>

Returns the number of unallocated granules on the disk in drive <drive>. 1 granule = 9 sectors of 23Ø4 bytes each. E.g. PRINT FREE(Ø)

GET <#buffer>,[record]

Reads [record] from DIRECT ACCESS file to <#buffer>. File must be OPEN. If [record] is not specified, next sequential numbered record will be read.

INPUT <#buffer>,<variable 1>[,variable 2],...

Transfers data from <#buffer> to variables <variable 1>, [variable 2],... in the order that the variables appear on the statement line.

KILL <"filename/ext[:d]">

Deletes file "filename/ext" from the disk. If drive [:d] is not specified, current default drive will be used.

LINE INPUT <#buffer>,<string variable>

Transfers a line of string data from <#buffer> to <string variable>. Data are transferred until "ENTER" character (ØD hex or 13 dec) is encountered.

LOAD <"filename[/ext][:d]">[,R]

Loads BASIC PROGRAM from disk file "filename" into memory. If filename extension [/ext] is not specified, /BAS is used. If [:d] is not specified then default drive is used. If [,R] is specified, program will be RUN immediately after it is loaded.

LOADM <"filename[/ext][:d]">[,offset]

Loads a MACHINE CODE program "filename" into memory. If filename extension [/ext] is not specified, /BIN is used. If drive [:d] is not specified, the default drive will be used. The program is loaded into memory at the load address it was saved from plus the offset (if supplied).

LOC <(buffer)>

Returns the record number of the record currently in <(buffer)>. E.g. A=LOC(1).

LOF <(buffer)>

Returns the last record number of the file associated with <(buffer)>. Used on DIRECT ACCESS files only.

LSET <field> = <data>

Transfers data from variable or constant <data> to <field> variable. Data will be left justified in <field> after transfer. E.g. LSET A$=B$

MERGE <"filename/ext[:d]">[,R]

Merges BASIC program file "filename" with BASIC program in memory. Program file to be merged must have been saved in ASCII format. (see SAVE) If drive [:d] is not specified, default drive will be used. If [,R] is specified, program will be RUN immediately after merging.

MKN$ <(data)>

Converts a numeric constant or variable <(data)> to a 5-byte string (see also CVN).

OPEN <"mode">,<#buffer>,<"filename[/ext][:d]">[,record length]

Opens a disk file "filename" for I/O through <#buffer>. Parameter <"mode"> is "I" for sequential access input, "O" for sequential access output, "D" for direct access input or output. parameter <#buffer> must be 1-15, and must not be already OPEN. If [/ext] is not specified , "/DAT" will be used. If [:d] is not specified, current default drive will be used. If [record length] is not specified, 256-byte records will be assumed (DIRECT ACCESS only).

PRINT <#buffer>,<data 1>[,data 2],...

Transfers data from constants or variables <data> to <#buffer>. Data are transferred in the order that the <data> are named on the statement line. Follows the same rules as for PRINT statement. E.g. PRINT #2,A$,B$;C;D

PRINT <#buffer>, USING <format>;<data 1>[data 2],...

Transfers data from constants or variables <data> to <#buffer>. Data are transferred in the order that it appears on the statement line. Follows the same rules as the PRINT USING statement. E.g. PRINT #1, USING "###.##";A

PUT <#buffer>[,record number]

Writes data in <#buffer> to [record number] of disk file associated with <#buffer>. If [record number] is not specified, record number of record currently in <#buffer> will be used. E.g. PUT #3, F

RENAME <"filename 1/ext[:d]"> TO <"filename 2/ext[:d]">

Renames file named "filename 1" to "filename 2". If [:d] is not specified, current default drive will be used. E.g. RENAME "MYPROG/BAS:2" TO "YOURPROG/BAS:2"

RSET <field> = <data>

Transfers data from constant or variable <data> to field variable <field>. Data are right justified in <field> after transfer. E.g. RSET A$=B$

RUN <"filename[/ext][:d]">

Loads and runs BASIC program file "filename". If "/ext" is not specified, the current default drive is used. E.g. RUN "GAME"

SAVE <"filename[/ext][:d]">[,A]

Saves the BASIC program in memory to disk file "filename". If "/ext" is not specified, "/BAS" will be used. If [:d] is not specified, the current default drive will be used. If ",A" is specified, the program will be saved in ASCII format.(I.E. not TOKENIZED) E.g. SAVE "MYPROG:2",A

SAVEM <"filename[/ext][:d]">,<first>,<last>,<execution>

Saves a machine code program in memory at address <first> through to <last> to disk file "filename". Entry point address <execution> will be saved with the file. If "/ext" is not specified, "/BIN" will be used. If [:d] is not specified, current default drive will be used. Parameters <first>,<last>, and <execution> must be decimal unless preceded with "&H" to indicate HEXADECIMAL. E.g. SAVEM "MYPROG",&H1ØØØ,&H14ØØ,&H12ØØ

UNLOAD [drive]

Closes all files currently open on [drive]. If [drive] is not specified, current default drive is assumed. E.g. UNLOAD Ø

```
VERIFY ON
VERIFY OFF
```

Turns ON or OFF the read-after-write verify option. Option remains in effect until changed or computer is RESET. After RESET verify option is OFF.

```
WRITE <#buffer>,<data 1>[,data 2],...
```

Transfers data from constants or variables <data> to <#buffer>. Data are transferred in the order that they are named on the statement line. E.g. WRITE #1,A,B,"STRING"

We hope that this booklet will be as educational to you as it has been to us as we put it together. Good Colour Computing.....

Bob Devries and Graham Butcher.

Disk Extended Color Basic Instruction Set. Error Messages.

Several new error messages are included in the Disk Basic ROM. They are listed here in alphabetical order with their meanings and corrective actions.

AE Already Exists

PROBLEM : File name specified as a new file name in a COPY or RENAME command already exists in the directory.

CORRECTIVE ACTION : Use a different file name or delete the existing file with the same name.

AO Already Open

PROBLEM : An attempt was made to open a file which was already open.

CORRECTIVE ACTION : Close the file. A file must be closed before it can be opened.

BR Bad Record Number

PROBLEM : The record specified does not exist in a direct-access file.

CORRECTIVE ACTION : Use the LOF function to determine the number of records in the file. Do not use a record number greater than that returned by the LOF function.

DF Disk Full

PROBLEM : There is not enough unallocated disk space remaining on the disk to complete the command.

CORRECTIVE ACTION : Either delete one or more files on the disk or use another disk with more unallocated space.

DN Drive Number

PROBLEM : A drive number has been specified which is not allowed.

CORRECTIVE ACTION : Use another drive number. Only drive numbers Ø, 1, 2, or 3 are allowed.

ER End of Record

PROBLEM : In a direct-access file, an attempt has been made to transfer data beyond the end of the record. The length of the record specified with the open statament must not be exceeded.

CORRECTIVE ACTION : Check the record length specification in the open statement. If the record length specification is correct, do not attempt to transfer more data than may be contained in a record of the length specified.

FD File Data Error

PROBLEM : An attempt has been made to transfer data between a file buffer and a variable of a different type. For example, if the data in the buffer is string data, and the variable is numeric, this error will occur.

CORRECTIVE ACTION : Use string variables to transfer data fields, and numeric variables for numeric fields.

FM File Mode Error

PROBLEM : An attempt has been made to access a file in a mode other than that for which it was opened. For example, an attempt to write data to a sequential access file which is open for input ("I" mode), will cause an FM error, as will using the LOAD command instead of LOADM for a BINARY file.

CORRECTIVE ACTION : Either close the file and re-open it for the correct mode, or change the statement in error to agree with the mode for which the file was opened.

FN File Name Error

PROBLEM : A file name has been specified which includes characters that are not allowed, or more than the maximum allowed number of characters.

CORRECTIVE ACTION : Correct the file name. A file name may contain any character except a slash ("/") or a period (".") or a colon (":"), and must be no more than 8 characters in length.

FO Field Overflow Error

PROBLEM : An attempt has been made to transfer more data to a field variable than the variable is defined to contain. For example, if a variable has been fielded as a 5 character field, and an attempt is made to transfer more than 5 characters of data to it, an FO error will occur.

CORRECTIVE ACTION : Either re-define the field variable, or re-define the data being transferred to it.

FS File Structure Error

PROBLEM : The disk granule table in the directory has been altered in such a way that the granules do not correspond correctly with the files in the directory.

CORRECTIVE ACTION : The disk must be re-formatted. Use the copy command to save as many files as possible, then re-format the disk.

IE Input past End of file

PROBLEM : An attempt has been made to read more data from a file than is contained in the file.

CORRECTIVE ACTION : Use the EOF and/or LOF functions to determine the file length. Do not attempt to read more data from the file than it contains.

IO Input/Output Error

PROBLEM : An error has occurred during a disk I/O. The error may be due to a number of problems such as CRC errors, no disk in the drive, etc. An unformatted disk or a disk not from the Color Computer will give this error.

CORRECTIVE ACTION : Check that the disk system is being used correctly.

NE Name Error

PROBLEM : The file name specified in a command was not found in the disk directory. The file does not exist.

CORRECTIVE ACTION : Either the file name was incorrectly specified, or the file may exist on another disk.

NO Not Open Error

PROBLEM : An attempt has been made to transfer data to or from a disk file which has not been opened with an open statement.

CORRECTIVE ACTION : Do not attempt to read or write to or from a disk file unless the file has been opened with an open statement.

OB Out of Buffer space Error

PROBLEM : Insufficient memory space has been allocated to the direct-access record buffer.

CORRECTIVE ACTION : Use the files statement to allocate more buffer space.

SE Set Error

PROBLEM : An attempt has been made to rset or lset data to a non-fielded variable. These functions may be used only with fielded variables.

CORRECTIVE ACTION : Use the rset and lset functions only with the variables which have been defined with the field statement.

VF Verification Error

PROBLEM : This error occurs during a disk write operation when the verify option is on, and the data read from the disk does not match that which was written.

CORRECTIVE ACTION : Re-issue the command. Many VF errors will not persist. If the problem persists, you may need to re-format the disk or use a new disk.

WP Write Protect Error

PROBLEM : An attempt has been made to write to a disk which is write protected.

CORRECTIVE ACTION : Either remove the write protect tab, or use another disk.

APPENDIX

This section lists the special features of Rainbow Bits 1.4 Disk Expanded Basic, which is supplied in locally built disk controllers.

BAUD RATE

The printer baud rate is now set to a default of 1200 baud. See Note 1.

STEPPING RATE

The rate at which the drives now step from track to track is now 6 milliseconds instead of the Tandy rate of 30 ms. This makes operation of the new slimline drives much quieter and faster. Do Not use these controllers with the old style Tandy drives.

DRIVE SELECTION

Four psuedo drives can be selected:-

```
Drive Ø (default) - side 1 of physical drive Ø.
Drive 1           - side 1 of physical drive 1.
Drive 2           - side 2 of physical drive Ø.
Drive 3           - side 2 of physical drive 1.
```

HEAD RESTORE

The power up sequence now sets up the Current Track Table so that the drive head is restored to track Ø on the first read or write of the disk. This reduces head banging problems.

DIRECTORY

The DIR command now results in a pause after one page of filenames has been printed on the screen. The listing may be continued by pressing any key except <BREAK> which will terminate the listing.

LIST

When listing a basic program the flow may now be halted by pressing any key instead of the usual <shift @> combination. The listing may be restarted by pressing any key except <BREAK> which terminates the listing.

VERIFY

The verify function now defaults to ON when the computer is turned on. If you wish to disable it, type "VERIFY OFF".

NEW COMMANDS

Eight new commands have been added for programming convenience.

AUTO <n,m>

Auto line numbering is invoked on input of <n> initial line number and <m> increment. If just AUTO is used alone it will default to 1Ø,1Ø.

DUMP

Text screen dump prints the entire contents of the text screen to the printer.

FLEX

If FLEX boot disk is in drive Ø it will start up the FLEX operating system.

OS9

If OS9 boot disk is in drive Ø it will start up the OS9 operating system. It is no longer necessary to have a separate OS9 boot disk. Note the OS9 command can also be used in the same way as the DOS command in the RADIO SHACK 1.1 DISK EXTENDED COLOR BASIC.

MON

Machine Language Monitor Routine. Starts up with a > prompt. Commands used are:-

F	- fill block of memory
G	- go to user program
E	- examine block of memory
O	- outputs m.l. program in disassembler form
X	- exits back to BASIC

Note all addresses must be supplied in HEXADECIMAL.

PON

Enables printer output. Any output which goes to the printer is also printed on the printer.

POF

Disables the printer. See PON.

RAM

Transfers all Basic ROMS to RAM and jumps into 64K mode.

RUNM"filename"

Loads and auto executes a machine code program from disk.

For those with special keyboards e.g. HJL with four extra (function) keys:-

F1	= Control
F2	= Upper/Lower case toggle
F3	= Auto repeat key
<shift>+F4	= Screen dump (text screen)

Software jump to "8Ø column crt" card is included if card is installed.

Note 1. Printer baud rate may be set to your personal preference when the disk controller is built.