Chapter 3

File Input/Output

Introduction to I/O

A computer is a communications device—it moves information from one place to another. The most common example is when the computer accepts information from a user through the keyboard, and presents information by displaying it on a screen. This process of communicating is referred to as input/output, and is often abbreviated as I/O.

The keyboard and screen are not the only parts of a computer system that can be involved in input/output. Other peripherals, such as a printer, tape recorder, disk drive, or modem, can also be used to transfer information. With a printer, you can get a permanent copy of a program listing or program results on paper. The Datassette recorder and the disk drive can be used to load and save programs and other kinds of information. By using a modem and a telephone, you can connect your computer to other similarly equipped computers.

Collectively, the keyboard, screen, printer, Datassette, disk drive, and modem are called *devices.* The transfer of information takes place between the computer and one of these devices. An important aspect of this information exchange is the direction in which the information is traveling. The direction describes whether information is being sent or being received, and is always described from the point of view of the computer. If the computer is receiving information from a device, the term *input* is used. When the computer is sending information to a device, it's termed *output.* Sometimes the terms *reading* and *writing* are used to mean *receiving* and *sending.*

With some devices, only one direction is possible. For example, the keyboard can be used only to input information. The keyboard is not capable of displaying information, and it would be ridiculous to try to make the computer send information to the keyboard. On the other hand, the printer is an output-only device. Information cannot be entered into the computer through the printer. Devices such as the Datassette, disk drive, and modem, however, support both input and output.

The purpose of this chapter is to show you how BASIC can be used to make the computer communicate with other

devices. BASIC's input statements are GET and INPUT, and its output statement is PRINT. With a few modifications, these same statements can be used with all the devices, not just the keyboard and screen. The next section introduces a new way to access the keyboard and screen, while the later sections show how these methods can be applied to the other devices.

Summary

- The process of the computer communicating is called *input/output,* abbreviated *I/O.* A typical example is when the computer receives information from the keyboard and transmits information to the screen.
- Those parts of the computer system which can be used for input or output—the keyboard, screen, printer, Datassette, disk drive, and modem—are called *devices.* The transfer of information takes place between the computer and a device.
- The direction of the transfer is described from the point of view of the computer. When the computer accepts information from a device, the term *input* is used. *Output* applies to information sent from the computer to a device.
- Some devices are capable of communicating in only one direction. The keyboard can be used only to input information, for instance, and the printer can be used only to output information. Other devices, including the Datassette, disk drive, and modem, can communicate in both directions.
- The rest of this chapter will show how to use variations of GET, INPUT, and PRINT to communicate with all devices, not just the keyboard and screen.

The OPEN and CLOSE Statements

To formally initiate communication with a device, you must create a *file* which is used for the actual input/output. Creating a file informs the computer that I/O is going to be performed. The file then acts as a sort of intermediary: Information to be sent to the device is first sent to the file; information coming from the device is first brought into the file to be retrieved by BASIC. Like a program, a file is more of a concept than a tangible reality.

For now, all that matters is that you realize a file is necessary in order to communicate with a device, and that the communication is done through the file. The details of how a

file works will become clearer as you explore its function and use.

A file is created by the OPEN statement, which has the following syntax:

OPEN ***file number, device number, command number, filename***

The **file number,** sometimes also called the *logical file number,* can range from 1 to 255. Numbers greater than 127 may have special significance and normally should not be used. The actual value is chosen by the programmer, but the most common is 1. The file number will later be used by the GET, INPUT, and PRINT statements.

Since it's possible for several devices to be active at the same time, the file number is needed to distinguish one currently active file from another. For example, a program could print a prompt on the screen, wait for a response on the keyboard, print a status message on the printer, and read in some information from the disk. For each device that's used, a new file must be created with a different file number. In the example above, the screen might have been opened as file 1, the keyboard as file 2, the printer as file 3, and the disk as file 4.

The **device number** is needed to specify which device is associated with the file. Each device is identified by its own number.

Number	Device
0	Keyboard
1	Datassette
2	RS-232 (modem)
3	Screen
4	Printer
5,6,7	Other devices
8	Disk drive

As long as the file is open, the device indicated by the device number will correspond with the file. All references to the file will be directed to the specified device.

The **command number** is also called the *secondary address.* It's not needed when using the keyboard or screen, and will be discussed in a later section.

The **filename** is needed when working with devices such as tape or disk drive. Like the command number, the filename will be dealt with later.

This leaves us with the simplest form of the OPEN statement:

OPEN *file number, device number*

Here's an example of how the statement would look in a program.

```
1Ø OPEN 1,Ø
```

This creates a file, designated as file 1, which uses device 0, the keyboard. To input information from file 1, the INPUT statement can now be used. However, since there might be several files open, to various devices, the computer will need to know from which file the input is expected. That's why the file number must be included as part of the INPUT statement, as shown by the next line.

INPUT# *file number, variable list*

In our example program, the INPUT statement would look like:

```
2Ø INPUT#1,N
```

The # symbol is considered a part of the keyword, so no space is allowed between the word *INPUT* and the symbol. A comma must be used after the file number. Using a semicolon causes a SYNTAX ERROR.

Add these two lines to lines 10 and 20 above, and run the short program to see how this special form of INPUT works.

```
3Ø PRINT
4Ø PRINT "YOU TYPED" N
```

Notice that there's no question mark printed as a prompt. The cursor just blinks, waiting for you to enter a number. When you type a number and press RETURN, the program echoes it back, as you'd expect. Now run it again, but this time enter some letters instead of a number. Instead of getting the REDO FROM START message, the program stops with a FILE DATA error. If you run the program once more and type two numbers separated by a comma, you'll not get the EXTRA IGNORED message—instead, just the first number appears. Also, if you examine the syntax for the new INPUT, you'll discover that no optional prompt message is allowed. As you can

see, INPUT used with a file is different from the INPUT with which you're already familiar.

The explanation of these differences is that the plain INPUT is a special form of INPUT#. Because INPUT is always used with the keyboard, features like prompt messages and error handling are helpful. However, since INPUT# may be used to fetch information from a device other than the keyboard, such as a data file on tape, features like prompt messages would not be appropriate or even desirable.

Let's try using the screen for output. Erase the old program by typing NEW, and enter this one.

```
10 OPEN 1,3 : REM 3 IS FOR SCREEN
20 PRINT#1,34
30 PRINT#1,"HAPPY BIRTHDAY"
40 PRINT#1,64,46
```

The # symbol is a part of the keyword, so this is one instance where the abbreviation for PRINT (?) cannot be used—you must type the full word PRINT#. As with INPUT#, a comma is needed after the file number. This first comma does not affect spacing. Run the program to see for yourself.

There's no difference between using PRINT and PRINT# when the file has been opened to the screen. If a program is going to be printing information only to this device, there's no sense in using PRINT# when the normal PRINT works just as well. The advantage in using the file method is that by changing the device number in the OPEN statement, all printing will be directed to a different device. By simply changing the 3 in the OPEN statement to 4, the messages will be printed on the printer instead of the screen (assuming a printer is connected). Use of the printer is discussed in greater detail in the next section.

Now let's consider using two open files at the same time. The next program uses file 1 for keyboard input, and file 2 for screen output.

```
10 OPEN 1,0 : REM KEYBOARD
20 OPEN 2,3 : REM SCREEN
30 PRINT#2,"WHAT IS YOUR NAME?";
40 INPUT#1,N$
50 PRINT#2
60 FOR K=1 TO 10
70 PRINT#2,N$
80 NEXT K
```

Because two files are open simultaneously, different file numbers *must* be used. If the second OPEN statement also used file 1, a FILE OPEN error would occur. Use this new line 20 with the program you just ran.

```
2Ø OPEN 1,3 : REM SCREEN
```

The error indicates that file 1 is already in use. A number (such as 1) cannot be used in another OPEN statement until the file currently using that number is finished with all input/output. At that time, the file should be *closed* with the CLOSE statement, which has a very simple syntax.

CLOSE ***file number***

The CLOSE statement formally terminates an input/output session with a device. The file created by the earlier OPEN no longer exists after CLOSE is executed, and the file number is free to use in opening another file. The following program shows how CLOSE is used.

```
1Ø OPEN 1,3 : REM SCREEN
2Ø PRINT#1,"WHAT IS YOUR NAME?";
3Ø CLOSE 1
4Ø OPEN 1,Ø : REM KEYBOARD
5Ø INPUT#1,N$
6Ø CLOSE 1
7Ø OPEN 1,3 : REM SCREEN
8Ø PRINT#1
9Ø FOR K=1 TO 1Ø:PRINT#1,N$:NEXT K
```

If you run this, you'll not see a FILE OPEN error. That's because the CLOSE statement closed file 1 after it was used.

At this point you may be wondering why it's even necessary to bother with a file at all. Instead of a file number after PRINT# and INPUT#, why not just use the device number? The answer lies in the fact that with some devices, other information besides a device number is needed. Input/output with the disk or cassette requires that a filename be specified. Entering the filename in an OPEN statement once, and then using the corresponding file number in each subsequent PRINT# or INPUT# statement is a lot easier than typing the filename every time those statements are used.

Another reason is that some devices (such as the disk drive) permit more than one file to be open at the same time. The computer could input information from two different disk

files. Since the device number for both these files would be 8, just using a device number in an INPUT statement would not be enough—the computer couldn't tell them apart. File structure using OPEN and CLOSE provides an organized means of communicating with *all* types of devices, and has proven to be a good method for general input/output.

There are a few other errors which can appear when using file input/output. You've already seen the FILE OPEN error when you tried to open an already opened file. The FILE NOT OPEN error occurs when you try to access a file that has *not* been opened. You'll see this error when you run this next program.

```
10 OPEN 1,3
20 PRINT#3 : REM FILE 3 NOT OPENED YET
```

The NOT OUTPUT FILE error happens when an attempt is made to output information to a device which can only handle input, such as the keyboard. Likewise, the NOT INPUT FILE error occurs when a program requests input from a device which is not capable of outputting information.

These kinds of errors can occur only when there is something wrong with a program's logic. They should not happen when the normal OPEN and CLOSE procedure is followed. The only two errors which denote some other type of problem are FILE DATA and DEVICE NOT PRESENT. The FILE DATA error (described as BAD DATA in Commodore literature) occurs when INPUT# is expecting a numeric value, but a string value is received instead. In some cases it's best to always use string variables with INPUT#, and the VAL function to determine if the value was a number. This prevents the program from crashing due to the FILE DATA error.

A DEVICE NOT PRESENT error appears when either the device is not connected to the computer or it's not turned on. A program has no way of preventing this type of error, so you just have to make sure that all peripherals are properly hooked up and have power before running a program that may use them.

A few more comments should be made about the CLOSE statement. It's good practice for a program to use CLOSE when it's finished sending or receiving information. This promotes good style and frees up a file for use later in the

program. When using tape or disk files, it's absolutely necessary to use CLOSE to terminate I/O.

A CLOSE of all files is performed as part of a CLR. Since CLR is part of the RUN, NEW, and LOAD commands, and also happens automatically when a program line is changed, any of these actions will close all open files.

Summary

- The OPEN statement is used to create a *file* through which input/output will be done. A file is the programming link between the computer and a device.
- The full syntax for the OPEN statement is the keyword OPEN, a file number, a comma, a device number, another comma, a command number, one more comma, and then a filename.
- The file number, also called the *logical file number,* is used to distinguish one active file from another. It's necessary because there may be several different files open at the same time.
- The file number is chosen by the programmer and may range from 1 to 255. File numbers greater than 127 are reserved for special purposes. File numbering usually starts at 1.
- The file number will later be used in the PRINT#, GET#, and INPUT# statements.
- The device number is used to identify the device associated with a file.
- A file can use only one device.
- The command number, also called the *secondary address,* and the filename are optional, and are covered in a later section.
- To make the INPUT statement read from a file instead of the keyboard, use INPUT#, the file number, a comma, and then a variable list, as in *INPUT#1,N.* No space is allowed between INPUT and the # symbol.
- The INPUT statement is actually a special form of INPUT# which is always set for keyboard input.
- Features like REDO FROM START and EXTRA IGNORED warning messages are not available with INPUT#. If INPUT# is expecting a numeric variable and gets characters, BASIC stops with a FILE DATA error. There's no warning if extra information is available. Also, INPUT# does not support an optional prompt string.

- These features are useful only when input is coming from the keyboard. Since INPUT# may get data from another device, it does not support these features.
- To make PRINT send output to a device other than the screen, use PRINT# followed by a file number, a comma, and then the information that is to be sent. No space is allowed between PRINT and #, and the ? abbreviation cannot be used.
- PRINT# to a file which has been opened to the screen is no different from using PRINT.
- Attempting to communicate with a file that's not been created by an OPEN causes the FILE NOT OPEN error. Attempting to open a file using a file number that is already in use causes the FILE OPEN error.
- When the computer is finished communicating with a device and the file is no longer needed, use the CLOSE statement to terminate the file. The syntax for this statement is the keyword CLOSE, followed by the file number.
- Once a file has been closed, it can be opened again for use with a different device.
- All files are automatically closed as part of CLR.
- The NOT OUTPUT FILE error occurs when you try to use PRINT# with a device that cannot handle output, such as the keyboard. The NOT INPUT FILE error occurs when you try to use INPUT# with a device that's not capable of generating input, such as the printer.
- The DEVICE NOT PRESENT error occurs when the computer tries to communicate with a peripheral that's either not hooked up or not turned on.

The Printer and the CMD Statement

A printer is an important part of a complete computer system because it can give you a permanent copy of information. Sending program output to a printer is as easy as using two statements.

```
10 OPEN 1,4 : REM PRINTER
20 PRINT#1,"MUSIC FILE LISTING"
```

All information being output by PRINT#1 will be sent to the printer. Of course, just plain PRINT will still print to the screen.

```
10 PRINT "TURN ON THE PRINTER"
20 PRINT "AND HIT ANY KEY"
30 GET K$:IF K$="" GOTO 30
40 OPEN 1,4 : REM PRINTER
50 PRINT#1,"MUSIC FILE LISTING"
60 PRINT "PRINTER IS WORKING"
```

You can also use a variable for the device number in the OPEN statement and let the person running the program decide where the information will be sent. This technique is demonstrated by the next program.

```
10 INPUT "OUTPUT DEVICE NUMBER";DN
20 OPEN 1,DN
30 PRINT#1,"MUSIC FILE LISTING"
```

The program user could enter either 4 for printer output or 3 for the screen. Using the OPEN statement like this means that you don't have to worry about where the printing will be sent. You can write the program just like any other, and as long as you use PRINT# wherever you would normally use PRINT, it will have the ability to send output to the printer, screen, or any other device.

There are just a few minor differences between printing on the screen and printing on paper. One obvious difference is that the printer will not know what to do with special CHR$ codes which move the cursor, change the current character color, or clear the screen. Printers usually do not have backspacing capability and cannot erase anything that's already been printed.

A more subtle difference has to do with the print formatting functions, POS and TAB. Both of these depend on the current location of the screen cursor. The POS function returns that value, and TAB uses it to determine how many spaces the cursor should be moved to the right. Since the screen cursor does not move when information is sent to the printer, these functions will not work correctly with printer output. This is not a major inconvenience, though, because the SPC function will still work and can be used to simulate TAB. The comma used for far spacing and printing in columns will also work differently for printer output.

The combination of OPEN and PRINT# can be very powerful. Nevertheless, there are some instances when it would be nice to send output to the printer which is not

program-generated. For example, it would be handy to be able to send a program listing to the printer. A program is much easier to read and debug when you can see it on paper, rather than just a screen at a time. For situations like this, the CMD statement is available.

CMD *file number*

The CMD statement causes all output normally sent to the screen to be sent to the designated file. This includes output from normal PRINT statements, INPUT prompt messages, disk and tape SEARCHING and LOADING messages, program listings, and even the READY. prompt. The file must, of course, be open when CMD is executed, or a FILE NOT OPEN error will occur. By using CMD, you can list a program on the printer by entering only three lines in immediate mode.

```
OPEN 1,4
CMD 1
LIST
```

This could also be used to print a disk directory. (Remember to load the directory before opening the file because LOAD closes all open files.) Any printing that defaults to the screen will now be sent to the file.

The syntax for CMD allows an optional comma and string to be placed after the file number. The string will be the first thing sent to the file when the CMD statement is executed. This is useful for such things as titling program listings.

```
LOAD "POEM"
OPEN 1,4:CMD 1,"POEM":LIST
```

CMD does not have to be used with just the printer. It can be used to send normal screen output to any device, such as the modem. Remember, CMD sends the output to a file; the device to which the information is sent depends on how the file was opened.

CMD stays in effect until the file is closed or an error occurs, in which case printing reverts back to the screen.

It's recommended that you send an empty print line before closing a CMD file, to help the computer terminate the I/O.

```
OPEN 1,4:CMD 1:REM START CMD
PRINT#1:CLOSE 1:REM STOP CMD
```

When CMD is used in a program, in effect it changes every PRINT into a PRINT#. This is useful when you have a

program which uses PRINT statements, but you want the output to be sent to a device different from the screen.

You might wonder why PRINT# has to be used at all if CMD is available. As it turns out, there are some reasons why it's better to use PRINT#. The major reason applies when you want your program to communicate with more than one device. If you use CMD to send output to one device and then switch to another, output may in some cases end up going to both. Also, CMD can sometimes turn itself off even when the file has not been closed.

PRINT# is more reliable and is usually the better choice. CMD is best used in the immediate mode when you want to do something that can't be done with PRINT#, such as listing a program to the printer. When using CMD in a program, make sure the program is sending output to only one device.

The remaining topics concerning the printer have to do with special modes of operation.

The file number in OPEN should normally be in the range 1–127. File numbers greater than 127 have a special meaning when used with the printer, because they instruct the computer to send a linefeed character after every carriage return. The linefeed character makes the printer advance the paper one line. Most printers do this automatically when they receive a carriage return. A printer which does not have this feature will keep printing on the same line. If your printer doesn't have the automatic linefeed feature, or if it does and you want the printer to use double-spacing, use a file number from 128 to 255.

If you're using a Commodore printer, the third parameter of the OPEN statement, the command number, can be used to select a character set. When no command is specified, the printer assumes that the command number is 0, which means that the uppercase and graphics character set will be used.

OPEN 1,4 : REM UPPERCASE/GRAPHICS

or

OPEN 1,4,0 : REM UPPERCASE/GRAPHICS

If you want the printer to use the uppercase/lowercase set, use 7 as the command number.

OPEN 1,4,7 : REM UPPERCASE/LOWERCASE

Remember, this applies only to Commodore printers.

Some printers support special operations modes. These may allow double-width printing, underlining, bold print, different type styles, dot graphics, and other features. They're invoked after a file has been opened by sending control codes to it. Control codes are sent as ASCII characters. For example, the statement PRINT#1,CHR$(14) makes a Commodore printer select the double-width mode, which means that characters are expanded to twice their normal width. Printing CHR$(15) turns the mode off. Other printers may use different numbers, so consult your manual for more information.

Sometimes printers require an *escape* character, abbreviated *ESC*, to be sent before the control code. The ESC code is character 27, so you would have to use PRINT#1,CHR$(27); CHR$(*control code*) to select the desired mode.

Summary

- To send program output to the printer, open a file to device 4 and use PRINT#.
- When sending output to the printer, the formatting functions POS and TAB, and comma spacing, will not work correctly.
- To send computer-generated output (such as a program listing) to the printer, open a file to device 4 and use the CMD statement. The statement's syntax is the keyword CMD, the file number, and an optional comma and string.
- The file must already be open, or the FILE NOT OPEN error will occur.
- If a string is included with the CMD statement, it will be sent to the file. A typical use of this feature is to title a program listing.
- When executed, CMD redirects all output that normally defaults to the screen to go to the file instead. This includes output from PRINT, as well as messages generated by the computer, such as the READY. prompt.
- CMD has no effect on printing sent to a file by PRINT#, even if the device associated with the file is the screen.
- When CMD is being used with a device other than the keyboard or screen, it's a good practice to send an empty print line to the file before closing it.
- CMD stays in effect until the file is closed or an error occurs. In either case, printing reverts back to the screen.
- It's best to use CMD only in situations where PRINT#

cannot be used, for it's not always reliable. This is especially true when the program uses multiple devices.

- File numbers greater than 127 make the computer send a linefeed after every carriage return. This is necessary if your printer does not automatically advance the paper. If your printer does, use a file number from 128 to 255 for double-spacing.
- When using a Commodore printer, the third parameter of the OPEN statement, the command number, can be used to select a character set. Use 0 for the uppercase/graphics set, which is the default set. Use 7 to switch to the uppercase/lowercase character set.
- Some printers support special modes of operation, such as double-width printing. These modes are enabled by sending control codes to the printer, either by printing a string or by using the CHR$ function. Sometimes a control code may have to be preceded by the ESC (escape) character, CHR$(27).

Files on Tape

Thus far you've probably used the Datassette only for storing and retrieving programs. This storage and retrieval is necessary because the computer can hold only one program in memory at a time—anything in memory is erased when the computer is turned off. It's much easier to load a program than it is to retype it every time you want to use it!

When a program is saved to tape, it's stored as a file consisting of program lines. That doesn't mean tape files can contain only program lines. They can contain other information, such as numbers and character strings, just as easily. Just as a program is lost when the computer is turned off, so is the data maintained by that program. You may want a program to save this data to tape for future reference.

An example is a word processing program which saves a document on tape for later changes and reprinting. Or a game might maintain a high-score file that would have to be periodically updated. A mailing list program is of little value if it can't save the names and addresses on tape or disk. A long and complex adventure game may have an option to stop play and continue later, in which case the current character status would have to be saved. These are all excellent applications of data files on tape.

The Datassette differs from other devices such as the keyboard and printer, since it can be used for both input *and* output. However, the Datassette can handle only one direction at a time, because the user controls the pressing of the RECORD button. So the direction has to be specified by the program when the file is opened. Another aspect of tape I/O is that filenames can be used. Both of these things can be handled by the OPEN statement.

OPEN *file number, device number, command number, filename*

This is the complete syntax of OPEN. The file number can range from 1 to 127. The device number for the Datassette is 1, which is the default value if none is specified. The command number tells the direction.

Number	Command
0	Input (read from Datassette)
1	Output (write to Datassette)

If no command number is given, the computer opens the file for input.

Finally, a filename, up to 16 characters long, can be specified. As when saving a program, a filename does not have to be specified. However, if none is given when opening a file for output, none can be specified when the file is later opened for input.

Here's an example of a file being opened for tape output.

OPEN 1,1,1,"TAPE FILE"

The file can later be opened for input by the following statement.

OPEN 1,1,0,"TAPE FILE"

Let's examine just how the information is stored and retrieved. The following program creates a tape file containing two items of information.

```
10 REM GAME HI SCORE DEMONSTRATION
20 N$="CHRIS":SC=12345
30 OPEN 1,1,1,"HI SCORE":REM TAPE OUTPUT
40 PRINT#1,N$:REM NAME
50 PRINT#1,SC:REM SCORE
60 CLOSE 1
70 END
```

When this program ends, a new file exists on tape. You can't load it as a program, but its contents can be retrieved by a program like this:

```
10 REM GAME HI SCORE DEMONSTRATION
20 OPEN 1,1,0,"HI SCORE":REM TAPE INPUT
30 INPUT#1,N$,SC:REM NAME AND SCORE
40 CLOSE 1
50 PRINT N$ " HAD A HIGH SCORE OF" SC
60 END
```

This is a rather simple example, but even so, it demonstrates a couple of important things. First, notice that the program which created the tape file used two PRINT# statements, instead of only one followed by a variable list. To understand why, let's consider exactly what's stored on the tape when PRINT# is used.

The first PRINT# writes the character string "CHRIS" to the file. Then, because there is no comma or semicolon at the end of the statement, a carriage return is written to the file. A carriage return is character 13, or CHR$(13). The second PRINT# writes the number *12345* to the file. Because BASIC prints positive numbers with one leading space and one trailing space, a total of seven characters will be written. The CHR$(13) for the second PRINT# would be the last character. The resulting file could be represented like this:

CHRISCHR$(13)(space)**12345**(space)CHR$(13)

Think of what happens when you use PRINT to print two values. The statement PRINT N$,SC will print the name CHRIS, then print a few spaces because of the comma, print the number 12345, and finally print the ending CHR$(13). Only one carriage return has been printed, but a lot of extra spaces have been inserted. This kind of file could be represented as follows:

CHRIS(several spaces)(another space)**12345**(space)CHR$(13)

Now think of what happens when INPUT is used to get two values. The statement INPUT N$,SC waits for the characters to be entered, then waits for either a comma or a carriage return, and finally waits for a number. (Remember that if you type only one value and press RETURN, the computer will print a ?? prompt and wait for the second value.) The same is true for PRINT# and INPUT#. The statement INPUT#1,N$,SC

expects either a comma or a CHR$(13) between the two values. The problem is that when the two values are printed with PRINT#1,N$,SC there won't be a comma or carriage return between the two values, just a whole lot of spaces. Here's what will happen:

1. INPUT#1,N$,SC will read all the characters up to the first CHR$(13) or comma. This means that N$ would be assigned the value "CHRIS 12345."
2. INPUT#1,N$,SC will then try to read a number to assign to SC. But since there's nothing more in the file, BASIC will stop with an error.

The description of the problem may be long, but the solution is simple. Either avoid using a variable list after PRINT# or put a CHR$(13) between each item. Each of the examples below sends the same characters to the file.

PRINT#1,N$:PRINT#1,SC (original example)

PRINT#1,N$;CHR$(13);SC
or
PRINT#1,N$ CHR$(13) SC

If you have a long list, it may be easier to assign the CHR$(13) to a string and use the string between each item, like this:

CR$=CHR$(13):PRINT#1,N$ CR$ SC CR$ A$ CR$ B$ CR$ C

These values can be retrieved by INPUT#1,N$,SC,A$,B$,C without any trouble.

The preceding examples have illustrated one other important point. The program reading the file must know how many values are to be read. An error such as STRING TOO LONG will occur when a program tries to read more data than is contained in a file. Furthermore, the program must know the order in which the values were written. If the first program had written the values with PRINT#1,SC:PRINT#1,N$ and the second program read them with INPUT#1,N$,SC (the variables are switched), there would definitely be a problem. The INPUT# would read the digits and assign " 12345 " to N$, and then stop with a FILE DATA error when it tried to read a number for SC and found characters instead.

In some cases, such as a game which keeps track of the top ten scores, the number of items never changes. However,

other files, like those maintained by a telephone directory program, may change size. How to know when to stop requesting input will be dealt with first.

There are three methods you can use to avoid running off the end of a file. One is to have the first item in the file, a number, indicate how many values follow. The example programs below use this technique. The first writes the names and scores to tape, while the second reads them from tape and displays them on the screen. Type in the first and run it; then enter the second and run it. (You need to rewind the tape to the beginning of the first file before pressing PLAY.)

```
100 OPEN 1,1,1,"HI SCORES"
110 PRINT#1,3:REM ITEM COUNT
120 PRINT#1,"BOB" CHR$(13) 14000
130 PRINT#1,"KEVIN" CHR$(13) 12000
140 PRINT#1,"ERIC" CHR$(13) 11000
150 CLOSE 1
160 END
```

```
100 OPEN 1,1,0,"HI SCORES"
110 INPUT#1,N:REM ITEM COUNT
120 FOR K=1 TO N
130 INPUT#1,N$,SC
140 PRINT N$,SC
150 NEXT K
160 CLOSE 1
170 END
```

(These programs only demonstrate the method, and are not intended to be practical applications.)

In some cases, a program may not know how many values will eventually be printed when the file is opened. Since no length number can be printed at the beginning of the file, the program must have another way to detect the file's end. This is similar to the problem of indicating the end of values read from DATA statements. You can use the same trick here as you would with READ and DATA—a flag. This write-and-read example uses *END* as a flag. Enter the first and run it; then type in and run the second.

```
100 OPEN 1,1,1,"HI SCORES"
110 PRINT#1,"MARK" CHR$(13) 504
120 PRINT#1,"JEAN" CHR$(13) 915
130 PRINT#1,"MARYBETH" CHR$(13) 412
140 PRINT#1,"KEN" CHR$(13) 755
150 PRINT#1,"END" CHR$(13) 0
160 CLOSE 1
170 END
```

```
100 OPEN 1,1,0,"HI SCORES"
110 INPUT#1,N$,SC
120 IF N$<>"END" THEN PRINT N$,SC:GOTO 110
130 CLOSE 1
140 END
```

The third method is to have the program monitor the value of ST, a reserved variable which holds status information. ST usually holds zero, but may contain another value when an error occurs or something important has happened. Bit 6 of ST will be set when the end of a file has been reached. The value associated with bit 6 is 64, so when a program sees that ST has the value 64 (or any nonzero value), it can stop requesting input. Take a look at this example for the technique's application. Enter and run these short programs as you did with the previous write/read samples.

```
100 OPEN 1,1,1,"HI SCORES"
110 PRINT#1,"MATTHEW" CHR$(13) 9
120 PRINT#1,"ANDREA" CHR$(13) 10
130 PRINT#1,"TODD" CHR$(13) 9
140 PRINT#1,"JEFFREY" CHR$(13) 8
150 CLOSE 1
160 END
```

```
100 OPEN 1,1,0,"HI SCORES"
110 INPUT#1,N$,SC
120 PRINT N$,SC
130 IF ST=0 GOTO 110
140 CLOSE 1
150 END
```

Notice that the end-of-file check was done *before* the printing in the previous method, but is done *after* the printing here.

Getting the correct values into the correct variables is simply a matter of taking care when writing the program.

Make sure that the variables in the PRINT# and INPUT# statements line up. The only thing that can cause trouble is when a null string is printed by PRINT#. The INPUT# statement will skip past a null string and read the next value instead.

For instance, the second program stops with an error at line 130.

```
100 N1$="GAIL":N3$="MAX":REM MISSING N2$
110 OPEN 1,1,1,"TAPE FILE"
120 PRINT#1,N1$
130 PRINT#1,N2$
140 PRINT#1,N3$
150 CLOSE 1
160 END
```

```
100 OPEN 1,1,0,"TAPE FILE"
110 INPUT#1,N1$:PRINT N1$
120 INPUT#1,N2$:PRINT N2$
130 INPUT#1,N3$:PRINT N3$
140 CLOSE 1
150 END
```

The program stopped because the null string was ignored—thus, N2$ was assigned the value intended for N3$. When INPUT# went looking for N3$, the file was empty. To avoid this potential pitfall, never print a null string. If you want INPUT# to retrieve a null string, have PRINT# print a space. INPUT# will not ignore the space, but when assigning the string variable, it will skip past all leading spaces. The effect is that the variable is assigned a null string.

An advanced feature of tape files is to use a command number of 2 in the OPEN statement. This has the same effect as using command 1, and prepares the file for output. The difference is that when the file is closed, it will be specially marked to indicate that it's the last file on the tape. This guarantees that when searching for a tape file, the computer will not look past the last file. Instead, the computer stops with a DEVICE NOT PRESENT error, which is equivalent to FILE NOT FOUND on the disk drive.

The direction of a tape file cannot be changed as long as it's open. If a tape file is opened for input and PRINT# is used, the NOT OUTPUT FILE error occurs. Likewise, the NOT

INPUT FILE error appears when INPUT# is used with a file that has been opened for output.

Before ending this section, it would be a good idea to stress the importance of using CLOSE. This statement is not so important when working with the keyboard or screen, but it's required when using tape or disk files. The Datassette can read only one file at a time. If a tape file opened for input is not closed, no other files can be opened until CLOSE is executed.

Much worse is when a CLOSE is accidentally omitted for a tape file that has been used for output. It's possible that some of the data will not be sent if the CLOSE is not performed. The file will be incomplete and the data lost—a program which later opens the file for input will be able to read only the first part, and may even crash because of an error.

In other words, a program should *always* perform CLOSE when communication with a tape file is completed.

Summary

- A tape file is not limited to storing only the lines of a program. Tape files can also contain data, in the form of numbers and strings, which can be used by a program.
- The Datassette can be used for both input and output, but can handle only one direction at a time. The direction (input or output) must be specified when the file is opened.
- Opening a tape file uses the full syntax of the OPEN statement: OPEN *file number, device number, command number, filename.*
- The device number for the Datassette is 1.
- A command number of 0 opens the file for input; 1 opens the file for output. Using 2 also opens the file for output, but marks the file as the last on the tape when it's closed. The command number is optional—by default, the file will be opened for input.
- A filename is optional but recommended. If none is specified when the file is created, none can later be used when the file is opened for reading.
- It's a good idea not to use a variable list after PRINT#. Commas in PRINT statements are used to format the screen display into columns, which is not necessary with tape files. Instead, output the items one at a time so that a carriage return will be printed after each. If you do use a variable list, print a CHR$(13) between each item.

- It is acceptable to use a variable list with INPUT#.
- A program should input variables in the same order they were written.
- The FILE DATA error occurs when a program tries to read character data into a numeric variable.
- If a program is expecting a string variable and a number is read instead, the string representation of the number, like that produced by the STR$ function, will be assigned to the variable.
- An error such as STRING TOO LONG appears when a program tries to read more information from a file than was written in the first place.
- There are three methods to avoid reading more data than exists in a file. One way is to use the first number to indicate how many items are in the file. This number can then be used to control a loop which reads the data.
- The other two methods work by detecting the end of the file while it's read. One way, often used with READ and DATA statements, is to use a flag, such as the word *END* or the number *−999*, as the last item written to the file.
- The third method is to stop when bit 6 of the status variable (ST) is set. ST has a value of 64 when the end of the file is reached, and has other nonzero values when I/O errors occur.
- INPUT# ignores null strings. If it reads a null string, it goes to the next item. To print a string so that it will be read as a null, print a single space.
- It's very important that CLOSE be executed when the computer is finished communicating with a tape file. An output file may not be written completely if this is not done.
- Only one file can be opened to the Datassette at a time. Once a file has been opened, its direction (input or output) cannot be changed.
- If a file is opened for input and PRINT# is used, the NOT OUTPUT FILE error occurs. If a file is opened for output and INPUT# is used, the NOT INPUT FILE error will occur.

Disk Files

The type of file just described for cassette is often referred to as a *sequential file*, one in which the information can be read back only in the order it was written. It's not possible to reread just one item in the file; you have to use OPEN again

and start at the beginning. With a sequential file it's also not possible to skip some items to read one later on. If a file has just been opened and you want to read only its last item, you have to read past all the other items first.

Sequential files are the simplest and most commonly used type of file. They are well-suited to the Datassette, because the computer cannot control the REWIND and F.FWD buttons. The nature of tape means that the Datassette can handle only sequential files.

But sequential files can also be used with the disk drive. There are a couple of differences in the OPEN statement, but otherwise the procedure is the same.

OPEN ***file number, device number, secondary address, string***

Any **file number** from 1 to 127 is acceptable. The **device number** for the disk drive is usually 8, although it can be changed to 9 when a second drive is used. The command number, also called the **secondary address,** serves a different purpose when using disk files. It will be explained in a moment. For now, just use 2. The **string** is required, and indicates the filename, type of file, and direction. The filename can be up to 16 characters, and is sometimes followed by the type of file (indicated by an *S* representing sequential) and perhaps a letter *R* or *W*, reading or writing. Commas are necessary to separate the filename, the type, and the direction. You *must* specify the type of file if you're writing to the disk—if you're only reading the file, you can omit the *S* designation. If no direction is indicated, the computer will open the file for reading. Here are some examples.

```
OPEN 1,8,2,"DOODLE,R" : REM READING, NO TYPE NEEDED
OPEN 1,8,2,"POEM,S,W" : REM WRITING, TYPE SPECIFIED
OPEN 1,8,2,"GAME" : REM READING ASSUMED
```

With minor changes, the high-score demonstration from the "Files on Tape" section can be used with a disk file. Type in both programs below. Load and run the first (make sure there's a disk in the drive). This creates a sequential file on disk. Now load and run the second. It reads the file from disk and displays the items on the screen.

```
100 OPEN 1,8,2,"HI SCORES,S,W"
110 PRINT#1,"KEVIN" CHR$(13) 80
120 PRINT#1,"CHRISTINE" CHR$(13) 90
130 PRINT#1,"ALISDAIR" CHR$(13) 91
```

```
140 PRINT#1,"LISA" CHR$(13) 87
150 CLOSE 1
160 END

100 OPEN 1,8,2,"HI SCORES"
110 INPUT#1,N$,SC
120 PRINT N$,SC
130 IF ST=0 GOTO 110
140 CLOSE 1
150 END
```

Examine the directory of the disk you just used. You'll notice that beside the filename HI SCORES is a three-letter code, normally PRG for a program file, but now SEQ. This is to remind you that it's a sequential file, contains data, and should not be loaded as a program.

The method of reading and writing a sequential file is pretty much the same for both tape and disk. A major difference between these two devices, though, is in the number of files that can be open at one time. While the Datassette can read or write only one file at a time, the disk drive can handle several files simultaneously. Files 1 and 2 could be open for reading, while file 3 would be open for writing. The only restriction when using multiple disk files is that each file must have a different secondary address. Secondary address numbers can range from 2 to 14.

```
100 REM PAYROLL PROGRAM
110 OPEN 1,8,2,"EMPLOYEES,R"
120 OPEN 2,8,3,"HOURS,R"
130 OPEN 3,8,4,"PAYCHECKS,S,W"
```

This program might read an employee's name and Social Security number from the first file, get the hours and wage information from the second, do whatever processing is necessary, and then send the results to the third. This last file would be used later to print paychecks. The example may not be practical, but it does illustrate how several disk files can be managed. Most applications don't require as many files open at the same time.

Another capability of the disk drive is that it can allow two open files to read from the same disk file.

```
100 OPEN 1,8,2,"WEATHER DATA"
```

(later in the program, when file 1 may still be open)

```
650 OPEN 2,8,3,"WEATHER DATA"
```

This will not work when a disk file is written. Once a disk file has been opened for output, it cannot be opened by a different file number for either input *or* output until it's closed.

A complication when using multiple files is that the status variable (ST) must serve for all files. ST always reflects the status of the most recently accessed file. This means that if a program reads from file 1, then from file 2, ST indicates the status of file 2—file 1's status information is lost. The solution is to assign ST to a temporary variable before accessing file 2. This variable can then be used to determine the first file's status. This next program illustrates the procedure.

```
100 REM READ ONE FILE TO ANOTHER
110 OPEN 1,8,2,"SOURCE,R"
120 OPEN 2,8,3,"DESTINATION,S,W"
130 INPUT#1,S$
140 A=ST
150 PRINT#2,S$
160 IF A=0 GOTO 130:REM BASED ON FILE 1 STATUS
170 CLOSE 1
180 CLOSE 2
190 END
```

There's another problem that you'll encounter when a disk file must be periodically updated. A file which maintains high scores, for instance, may be read when a game is finished to compare the current score against the high score. If the player did well and the high score file has to be modified, the old file has to be erased and a new one written using the same filename. This poses no problem with tape files since the tape can be positioned to start at the beginning of the old file. The disk drive, however, won't let you write a file with a filename already in use. A different solution must be found.

Secondary addresses from 2 to 14 are available for normal disk file handling. Addresses 0 and 1 are reserved for LOAD and SAVE format files. The number *15* is used to send commands to the disk drive. One of these commands is called **Scratch,** and informs the drive that it should erase a file on the disk. For example, the file "POEM" could be erased by entering this line in immediate mode.

```
OPEN 1,8,15:PRINT#1,"S0:POEM":CLOSE 1
```

This is one case where no string has to be included as part of the OPEN statement. The secondary address (15) indicates that the *command channel* will be referenced, so no filename or type of file is needed. The command is sent as a one-letter code, followed by a zero for the drive number, a colon, and then the filename. You can also scratch files with this procedure from within a program.

If a program is going to open a disk file for output, and there's a chance the filename may already exist, the program should first send the Scratch command. (No error occurs if an attempt to scratch a nonexistent file is made.) Here's an example of scratching a file before writing a revised version of it to disk.

```
100 OPEN 1,8,15
110 PRINT#1,"S0:HI SCORES"
120 CLOSE 1
130 OPEN 1,8,2,"HI SCORES,S,W"
140 REM CONTINUE WITH REST OF PROGRAM
```

Perhaps you've used the *at (@)* character to save and replace a file. Assuming there's a file on disk named GAME, the statement SAVE "@0:GAME",8 scratches the old file GAME, then saves the current BASIC program using that filename. This replaces the file on the disk with the one in memory.

Do not use the @ character. This feature may not always work correctly. In some cases, the disk drive may even tamper with other files on the disk. Even though using the @ character to save and replace may seem very convenient, my advice is to avoid it. Use the Scratch (S0:) method instead.

Other useful disk drive commands are New, Rename, and Validate. **New** is used to format a disk and clear its directory. The full format of the command is "N0:*diskname, identification*", where the disk name is up to 16 characters and the disk identification 2 characters long. Make sure that no two disks are given the same disk identification. This precaution insures that there's no confusion between the source and destination disks when copying files from one to the other.

If the identification is omitted, all the files will be erased and the new disk name will be used, but the disk will *not* be reformatted. This can come in handy when you want to use the disk for a new project, but it's already been formatted. Formatting really needs to be done just once. Skipping the

process saves time. Take a look at the following lines; they show how to format a disk, as well as how to create a new directory only.

OPEN 1,8,15
PRINT#1,"N0:GAMES,01" (format disk and create new directory)
or

PRINT#1,"N0:GAMES" (create new directory only)
CLOSE 1

Rename lets you change the name of a file without disturbing its contents. Here's the proper syntax:

PRINT#1,"R0:*new filename = old filename*"

This example shows how a file named DOODLE can be renamed as POEM (assuming the OPEN has already been executed).

PRINT#1,"R0:POEM=DOODLE"

When using Rename, make sure the new filename is not already in use on the disk.

Validate is used to recalculate the number of free blocks on a disk. If some blocks have been allocated for files, but have not been used, they'll be made available after the Validate command is sent. This command also deletes any files which were never properly closed. Such files are marked in the directory with an asterisk (*) before the three-letter file type. Validate does not alter any properly closed files. If you validate periodically, you may be able to get a few more free blocks, letting you squeeze more on the disk.

The secondary address of 15 refers to the command channel when PRINT# is used. INPUT# can be used with this channel to get error messages. However, BASIC reports only a few I/O error messages, such as FILE NOT FOUND and DEVICE NOT PRESENT. When an error occurs, the disk drive can provide more information if you run the following short program.

```
10 OPEN 1,8,15
20 INPUT#1,A,B$,C,D
30 CLOSE 1
40 PRINT A;B$;C;D
```

Normally, the message 0 OK 0 0 will appear, indicating all is well.

If you tried to save a program using a filename already in

use, BASIC would not print an error message, but the drive light would flash. Run the above program, and you'll see the message 63 FILE EXISTS 0 0. Error number 63 tells you that an attempt was made to create a new disk file with an existing filename. Other error numbers and their descriptions can be found in the disk drive user's manual.

(The status numbers returned in variables C and D are for advanced applications, and can be ignored.)

There are a couple of differences between normal disk files (which use secondary addresses 2–14) and a file opened to the command/error channel (secondary address of 15). No direction is specified when the file is opened, and it's possible to use both PRINT# and INPUT# without having to close and reopen the file.

The other difference is that when the command/error channel is closed, the disk drive closes all the other disk files as well. BASIC may think that the files are still open, but an error will occur when the program tries to read from them. Thus, if a program is going to use the command/error channel while other disk files are open, it's best to open the channel during initialization and leave it open until the program ends. Many people set aside file 15 for this purpose.

Sometimes it would be handy if a program could read the disk directory. This is easy to do, but several things must be done differently. First, the OPEN statement should use "$0:" (or just "$") as the filename. Also, because of the structure of the directory file, INPUT# cannot be used. The file has to be read a character at a time. Fortunately, this can be done by the GET# statement.

GET#'s syntax is the keyword GET#, followed by a file number, a comma, and then a variable list. Something like GET#1,A$ retrieves the next character from the file and assigns it to A$. Only string variables should be used with GET# to avoid the FILE DATA error.

The following demonstration shows how to read a sequential file character by character.

```
10 OPEN 1,8,2,"HI SCORES"
20 GET#1,A$:PRINT A$;:IF ST=0 GOTO 20
30 CLOSE 1
```

The directory file contains characters for block lengths and filenames. The first two characters in the file can be ig-

nored. The first line in the file is the directory header, giving the disk name, identification, and DOS (Disk Operating System) version. Several lines follow, one for each entry. Again, the first two characters of each line can be ignored. The next two form the low-byte and high-byte number which indicates the file's block length. A few spaces precede the opening quotation mark, which begins the filename. After the filename and the closing quotation mark, there are several more spaces and then the three-letter file type. Each line ends with a CHR$(0), interpreted by GET# as a null string. This format repeats for each line in the directory. The block length for the last line shows the number of free blocks on the disk, with the filename BLOCKS FREE.

The end of the directory file is detected by checking ST after reading the block length. This program illustrates how it's done.

Program 3-1. Directory Reader

For error-free program entry, be sure to use "The Automatic Proofreader," Appendix D.

```
10 OPEN 1,8,0,"$":GET#1,S$,S$                    :rem 204
20 GET#1,S$,S$,L$,H$:IF ST THEN CLOSE 1:END
                                                 :rem 103
30 PRINT ASC(L$+CHR$(0))+256*ASC(H$+CHR$(0));
                                                 :rem 163
40 GET#1,S$:IF S$>"" THEN PRINT S$;:GOTO 40:rem 66
50 PRINT:GOTO 20                                 :rem 199
```

The directory file can only be read and cannot be opened for output.

A handy trick with reading the directory is to put a filename after the $0:, as in "$0:POEM". This makes the disk drive search the directory for the filename POEM and list it if it's on the disk. If there's no file named POEM, there'll be no directory lines read after the header. This is useful when you want a program to check if a file already exists. If the directory is opened with the filename after the $0: and no filename lines are read, the file isn't on the disk.

A variation on this technique is to use *wild cards* in the filename. If you use a question mark (?) as one of the characters in the filename, the disk drive will ignore that character position when searching for filenames. Let's say a disk contains the following files.

DOODLE
GAME1PLAYER
GAME2PLAYER
POEM
GMINUET
AMINUET
SONATA1
SONATA22
SONATINA1
SONATINA3
DMINUET

Using the string "$0:?MINUET" would read only the three minuet files, since the ? forces the disk to ignore the *G, A,* and *D* characters in the files. If you used the string "$0:GAME?PLAYER", both GAME1PLAYER and GAME2PLAYER would be read. If necessary, several question marks can be used in the same filename.

A second wild card is the asterisk (*). While the question mark can take the place of only one character, the asterisk can be substituted for several. An asterisk at the end of a filename makes the disk drive search for all filenames which begin with the letters up to the asterisk. All characters at the position of the asterisk and after are ignored.

In the example directory, if "$0:SON*" were used as the filename, all files starting with the letters *SON* would be returned in the directory, four altogether. With "$0:SONATA*", only the filenames SONATA1 and SONATA22 would be read.

A combination of the ? and * wild card characters can be used in the same filename.

When loading a program or opening a file for reading, wild cards within the filename make the computer access the *first file* on the disk which matches. This can come in handy if you can't remember or don't want to type the full name of a program. LOAD "PROG*",8, for instance, loads the first filename on the disk which has PROG as its first four characters.

Wild cards can be very powerful if you choose filenames to accommodate them. Just be careful that the filenames referenced with wild cards are the ones you wanted. This is especially true when using wild cards with the Scratch command. It's quite easy to erase too many files—more filenames may fit the wild card pattern than expected. In fact, it may be

best to avoid using wild cards with Scratch. At least check which files will be scratched beforehand by reading the directory with the wild card pattern.

As you've seen, sequential files are identified in the directory by the letters *SEQ*. The disk drive also supports another three-letter code, *USR*, for the same type of file. A USR file is handled just like a sequential file; the only difference is in the directory listing. Since some system utilities such as the Editor use SEQ files, you may want to use the USR type in your programs to help distinguish program data files from system files.

The full syntax for the string in the disk OPEN statement is:

filename, optional type, optional mode

The type can be SEQ, USR, PRG, or REL. These last two types will be discussed in a moment. If no type is specified, the default SEQ is used. (Remember that the type is *not* optional if you're writing. It must be included.) The mode tells the direction, and can be Read or Write. Default is Read.

Both the type and mode permit single-letter abbreviations; the type can be designated by S, U, P, or R, and the mode can be R or W. A file identified by the letters USR in the directory could be opened for output with:

OPEN 1,8,2,"HIGH SCORES,U,W"

From there, the handling would be just as if the file were of the type SEQ.

File type PRG is reserved for programs stored and retrieved using the SAVE and LOAD commands. It's possible to open these files and read them, but the reading has to be done one character at a time. Another problem is that due to the way the program is stored, the file may contain several nonprinting characters. The best way to examine a program file is to print the ASCII value of each character with something like this routine.

```
10 OPEN 1,8,2,"PROGRAM,P":REM DEFAULTS TO READING
20 GET#1,S$
30 PRINT ASC(S$+CHR$(0))
40 IF ST=0 GOTO 20
50 CLOSE 1
```

The +CHR$(0) in the ASC function (line 30 above) is needed because the file may contain several CHR$(0) characters. A CHR$(0) is interpreted by GET# as a null string, which

when used as the argument of the ASC function makes BASIC stop with an error. This can be averted by appending a CHR$(0) to the argument. If the string does contain characters, the extra CHR$(0) at the end won't affect the ASC function because ASC looks only at the first character. If the string is null, it becomes CHR$(0), which makes ASC return the 0, the correct value.

Reading a PRG file is useful only for advanced applications, such as programs which search for all variable names in a program, renumber a program, or compact a program by removing extra spaces and combining lines.

The one remaining type of file supported by the disk drive is the REL, or *relative,* file. Unlike a sequential file, this type of file makes it possible to back up to an earlier part of the file without having to use OPEN again, and to skip past items in the file without reading or writing them. A relative file is also known as a *random access* file because the reading or writing of items can be done arbitrarily, without having to follow any order.

Such a file structure is useful when a file contains several entries, only a few of which may have to be changed at any time. If it's necessary to read an account for a customer near the end of the file, and update the account of a customer near the beginning, these read and write operations can be performed quickly without disturbing the rest. REL files make it easy to handle large amounts of information. In fact, it's possible for a REL file to contain more data than could be held in memory at one time.

Unfortunately, Commodore 64 BASIC does not directly support relative files; it's a very complicated subject, far beyond even the nature of these discussions of advanced BASIC.

Summary

- In a sequential file, data must be read in the same order as it was written. It's not possible for reading to back up or skip ahead.
- With a few changes to the OPEN statement, sequential files can be used with disks. The syntax to open a disk file is OPEN *file number, device number, secondary address, string.*
- File numbers should range from 1 to 127. The device number for the disk drive is usually 8, but can be 9. Secondary ad-

dresses 0 and 1 are reserved for use by the LOAD and SAVE commands. The numbers 2–14 are used for data files. Each open file must have a different secondary address. A secondary address of 15 is used to access the command and error channel of the disk drive.

- The string consists of a filename, a file type, and a direction. The filename is required and can be up to 16 characters. The file type is optional when reading, mandatory when writing. File type can be SEQ, USR, PRG, or REL (abbreviated as S, U, P, and R, respectively). If no file type is specified, SEQ is assumed. The direction is optional, and can be reading or writing (or R or W). If no direction is given, the file is opened for reading.
- Unlike the Datassette, several different files can be opened to the disk drive simultaneously. Some can be set for input, others for output.
- Two or more files can be opened for input from the same disk file at the same time. However, once a disk file has been opened for output, it cannot be read or written by any other file until it is closed.
- The disk drive command and error channel is accessed by opening a file with a secondary address of 15 and no string, as in OPEN 1,8,15. No filename or direction has to be specified.
- While the file is open, PRINT# can be used to send commands to the drive, and INPUT# can be used to get status information. Unlike normal disk files, the file does not have to be closed and reopened to switch between input and output.
- The Scratch command is used to erase one or more files on a disk. The command string is "S0:*filename*". The *S* represents the Scratch command, the *0* is the drive number, and the filename indicates which file is to be erased. *If the filename contains wild card characters, more than one file may be erased.*
- Do not use the @ character in the filename of an OPEN or SAVE to replace and save a file on a disk. This feature is not reliable and can damage the contents of a disk.
- The New command is used to format a disk and clear the directory. The command string is "N0:*diskname,id*".
- The Rename command is used to change the name of a file. The command string is "R0:*new filename=old filename*". The contents of the file are not altered in any way.

- Validate is used to make previously allocated blocks not used by files available on the disk. Files that were never properly closed (indicated by an asterisk before the file type) will be erased. Properly closed files will be left alone. After the disk has been validated, there may be a few more free blocks.
- BASIC reports only the errors FILE NOT FOUND and DEVICE NOT PRESENT, and sometimes doesn't report an error at all. A flashing light on the disk drive may be the only clue that an error has occurred. More specific error messages are available from the disk drive. Assuming that file 1 has been opened to the error channel, the statement INPUT#1,A,B$,C,D will read the error number into A, the error message into B$, and other values for advanced applications into C and D. When no error has occurred, the error channel will return the values 0, OK, 0, 0.
- The GET# statement is used to input single characters from a file. The syntax is the keyword GET#, the file number, a comma, and a variable list. Only string variables should be used with GET#.
- The directory can be read by a program by using OPEN with the filename "$0:" and no file type or direction. The format of the directory file requires that it be read one character at a time, using GET#.
- The directory file will contain all the filenames on the disk when the string used with OPEN is $0:. To make the disk drive search for a specific filename in the directory, put the filename at the end of the string, as in "$0:DOODLE". The corresponding filename line will be read if the file exists. If not, the directory file will contain only the disk header and free blocks lines.
- The wild card characters (? and *) can be used in the filename after $0: to list all filenames which match a certain pattern.
- When wild cards are used with LOAD or OPEN for input, the computer will access the first file on the disk that has a filename matching the wild card pattern.
- The file type USR is identical to the file type SEQ. The different name can be useful to help distinguish between sequential files used for different purposes.
- The file type PRG is used for LOAD and SAVE type files, including BASIC and machine language programs. In ad-

vanced applications, a PRG file can be read or written one character at a time.
- Another type of file is the *relative* file, identified as REL. In a relative file, reading or writing can jump back or ahead. No particular order must be followed. Relative files, although supported by the disk drive, are not supported by Commodore 64 BASIC.

The Modem and Other RS-232 Devices

RS-232 refers to an agreement among manufacturers of electronic equipment (Electronic Industries Association) which defines how devices communicate. Its specifications include such things as power levels, which connector pins serve which functions, and so on. This insures that products from different manufacturers will be more or less compatible. Like ASCII, it's one of the few standards in the computer industry.

The RS-232 standard applies to *serial communication.* The usual method of transfer is parallel communication, in which information is sent a byte at a time. All eight bits of each byte are sent simultaneously. In serial communication, however, information is sent a bit at a time. To send a full byte, the eight bits have to be sent separately, one after another. Devices which communicate serially include modems and some printers.

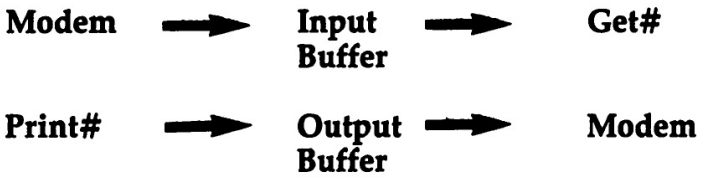
A *modem* converts bits into tones which can be sent over telephone lines to another computer. The receiving computer must also be connected to a modem, which then converts the tones back into bits. The general term for this is *telecommunications.* The process of changing bits into tones is called *modulation,* and the process of converting the tones back into bits is called *demodulation.* In fact, the word *modem* is derived from the words *MOdulate* and *DEModulate.*

In telecommunications, *transmit* is often used to mean *send.* The modem is different from most other devices because it can both transmit and receive information at the same time. But a computer can handle only one character at a time. If a character comes in while the computer is busy sending another, the computer must remember that incoming character and deal with it later. It's also possible for the computer to output characters faster than the modem can transmit them. To avoid losing incoming characters or to hold outgoing characters until the modem can take them, a *buffer* has to be

used. A buffer is a temporary holding place for characters. When using the modem, the computer needs two buffers, one for input and another for output.

To better understand the concept of a buffer, let's consider an example using an input buffer. Characters received are first stored in the input buffer, in the order in which they are received. Every time GET# executes, it retrieves the next character from the buffer. If the buffer is empty, GET# returns with the null string. The process could be illustrated like this:

Figure 3-1. Buffer



The following steps show how the method might work in practice.

1. The buffer starts out empty.

   _ _ _ _ _

2. The modem receives an *M* character and places it in the buffer.

   M _ _ _ _

3. The GET# statement retrieves the first character from the buffer (M). The buffer is emptied.

   _ _ _ _ _

4. The modem receives the character *A* and places it in the buffer.

   A _ _ _ _

5. The modem receives another letter, *R*, before the computer has retrieved the previous character. That's okay, since the *R* can be placed in the buffer after the *A*.

   A R _ _ _

6. GET# draws off the first character in the buffer (A). The *R* is now the first character in the buffer.

   R _ _ _ _

7. GET# gets the *R* and the buffer is once again empty.

   _ _ _ _ _

8. GET# tries to get another character, but since the buffer is now empty, GET# returns a null string. The program keeps looping until it gets something other than a null string.
9. The modem eventually receives a letter *K* and puts it in the buffer.

   K _ _ _ _

10. The next time GET# executes, it returns with *K* instead of a null string. The buffer is empty.

    _ _ _ _ _

Two important observations should be made. The modem received the characters in the order *M–A–R–K*, and the program (with GET#) retrieved the characters in that order. Even though the computer wasn't always able to retrieve each character as it came in, the buffer insured that no characters were lost.

The output buffer works in much the same way. Characters sent by PRINT# are held in the buffer until the modem is ready to transmit them.

Here's the syntax for the OPEN statement when used with an RS-232 device.

OPEN ***file number, device number, secondary address, string***

File numbers can range from 1 to 127. Numbers from 128 to 255 cause the computer to send a linefeed (CHR$(10)) after every carriage return. The linefeed character is required for printers which do not automatically advance the paper. The **device number** for an RS-232 device is 2. The **secondary address** is not used and should be 0. The **string** can be one or two characters. The first character is a control value, and the second is an optional command value.

The **control character** sets the number of stop bits, the word length, and the baud rate. Remember that each character is sent as a sequence of bits. Stop bits help the receiving modem detect the end of each character. The Commodore 64 can support either one or two stop bits, depending on the value of bit 7 in the control character.

Bit 7	Value	Effect
0	0	1 stop bit
1	128	2 stop bits

The *word length* determines how many bits are transmitted for each character, and can range from five to eight bits. Bits 6 and 5 of the control character establish the word length. For most purposes, eight bits will be just fine. When communicating with some modems or printers, especially older ones, fewer bits may be required. The device owner's manual should tell you if less than eight bits are to be used.

Bits 6	5	Value	Effect
0	0	0	8-bit word
0	1	32	7-bit word
1	0	64	6-bit word
1	1	96	5-bit word

The *baud rate* determines how quickly the bits are transmitted. The most common rate is 300 baud (roughly 300 bits per second), which is equivalent to about 30 characters per second. Faster baud rates (600 or 1200, for instance) require higher quality circuitry and make devices which are capable of such rates more expensive. Older printers and modems may even operate at 110 baud. Bits 3 through 0 of the control character should be set according to the baud rate of the device.

Bits 3	2	1	0	Value	Effect
0	0	0	1	1	50 baud
0	0	1	0	2	75 baud
0	0	1	1	3	110 baud
0	1	0	0	4	134.5 baud
0	1	0	1	5	150 baud
0	1	1	0	6	300 baud
0	1	1	1	7	600 baud
1	0	0	0	8	1200 baud

Here's the formula for calculating the value of the control character. Place the bit values from the tables above in the appropriate places in the formula.

Control character = CHR$(stop bits + word length + baud rate)

A typical setting is one stop bit, eight-bit word length, and 300 baud.

Control character = CHR$(0 + 0 + 6) = CHR$(6)

The result would be used in the OPEN statement like this:

```
OPEN 1,2,0,CHR$(6)
```

The **command character** sets the parity, duplex, and handshaking. *Parity* is used to check for errors in transmission. If there's considerable static on the telephone line, a bit may be accidentally switched in value, resulting in garbled communications. Parity checking can detect an error like this. When parity checking is selected, it can be odd or even. The correct setting depends on the setting of the device at the other end of the link. Bits 7 through 5 control the parity.

Bits 7	6	5	Value	Effect
0	0	0	0	No parity
0	0	1	32	Odd parity
0	1	1	96	Even parity

When using a word length of eight bits, parity checking is not available—no parity should be selected.

Duplex determines whether both modems can transmit at the same time or if they have to take turns. The usual value is full duplex. The other setting, half duplex, is rarely used. Again, this setting depends on the receiving modem. If half duplex is desired, bit 4 of the command character should be set (to 1). In most cases, especially with newer modems, full duplex is available, so bit 4 should be clear, or 0.

Bit 4	Value	Effect
0	0	Full duplex
1	16	Half duplex

Handshaking refers to how the transmitting and receiving devices recognize and establish communication between each other. Handshaking is controlled by bit 0, which for most applications should be clear, or 0.

As you can see, the command character is not as important as the control character. If the sum of the command values is zero, the command character does not have to be used. If a different value is needed, the command character is concatenated to the control character.

This next OPEN statement shows how that's done to select half duplex (bit value of 16).

```
Command character = CHR$(16)
OPEN 1,2,0,CHR$(6)+CHR$(16) : REM SELECT HALF DUPLEX
```

When the OPEN statement is executed, it creates input and output buffers of 256 bytes each. This allotment of 512 bytes comes from free memory. It forces an automatic CLR, which erases all previous variables and closes all other open files. Thus, if a program is going to use an RS-232 device, the OPEN should come before any variable assignments, array dimensions, function definitions, or other OPEN statements.

The 512 bytes are restored when the RS-232 file is closed. CLOSE causes another CLR, so the CLOSE should be the last thing in the program.

The variable ST has a different meaning when used with an RS-232 device. The last three bits are set to indicate errors.

Bit	Value (set)	Error when bit set
0	1	Parity error
1	2	Framing error
2	4	Receiver buffer overflowed

A parity error occurs when a bit has been switched in value, probably due to a bad connection. Don't bother to check this bit when using a word length of eight bits.

IF ST AND1 THEN parity error

(Note: Due to a quirk in BASIC, there must be a space between ST and AND.)

A framing error also indicates a problem in receiving a character.

IF ST AND2 THEN framing error

The buffer OVERFLOW error occurs when the input buffer is full (contains 256 bytes) and another character is received before the program retrieves a character. This can happen when the program handling the input and output is too slow for the baud rate used. A BASIC program can run only so fast. If the program has to do a lot of processing and a fast baud rate such as 600 or 1200 is used, this error may occur. The solution is to write the program in machine language.

IF ST AND4 THEN buffer overflowed

Another point about the ST variable is that it can be read only once after an input/output statement. After it's read, it's automatically set to zero. If you need to read it more than once, assign it to a temporary variable.

RS-232 communication may seem complicated in its description, but it can be very easy to implement. To show just

how easy it is, take a look at Program 3-2, "Modem." It's a simple program which makes your computer and modem act as a terminal, allowing you to communicate with another computer, perhaps one running a bulletin board system. The program is set for 300 baud and cannot handle faster rates. To use this program, you must have a modem connected to your Commodore 64 (and another computer with a modem to receive your transmission).

After typing in and running Program 3-2, make sure your modem is properly connected. Access the receiving computer as you would normally, by dialing its telephone number. Although the program establishes full duplex, you can switch your modem to half duplex (if it has that feature) to see what you type on the screen.

Program 3-2. Modem

For error-free program entry, be sure to use "The Automatic Proofreader," Appendix D.

```
10 OPEN 1,2,0,CHR$(6):POKE 56577,0          :rem 5
20 DIM T%(255),R%(255):FOR K=1 TO 255:T%(K)=K:R%(K
   )=K:NEXT                                  :rem 112
30 FOR K=65 TO 90:T%(K)=K+32:R%(K)=K+128:NEXT:FOR
   {SPACE}K=97 TO 122:R%(K)=K-32:NEXT        :rem 245
40 FOR K=193 TO 218:T%(K)=K-128:NEXT:T%(20)=127:R%
   (127)=20:PRINT "{CLR}{N}"                 :rem 234
50 GET A$:IF A$>"" THEN PRINT#1,CHR$(T%(ASC(A$)));
                                             :rem 122
60 GET#1,A$:IF A$>"" THEN PRINT CHR$(R%(ASC(A$)));
                                             :rem 121
70 GOTO 50                                    :rem 5
```

A few comments about this program might be helpful. Lowercase is obtained by using the second character set. Unfortunately, this means that some character translation has to be done to convert between ASCII codes and character set numbers. The translation process works like this:

Receive	Print on screen as
0–64	0–64 (punctuation)
65–90	193–218 (uppercase)
91–96	91–96
97–122	65–90 (lowercase)
123–126	123–126
127	148 (backspace)

Keyboard	Transmit
0–64	0–64 (punctuation)
65–90	97–122 (lowercase)
91–96	91–96
148	127 (backspace)
193–218	65–90 (uppercase)

The fastest way to translate is with arrays. Arrays T% and R% have been set up for this purpose (line 20 in Program 3-2).

There are a couple of restrictions in using an RS-232 device. First, only one such device can be used at a time. Second, an RS-232 device should not be used while another peripheral, such as the Datassette or disk drive, is in use.

The last topic concerns receiving a CHR$(0). If the modem receives a CHR$(0), it stores it in the buffer properly, but the GET# statement interprets it as a null string. This means a null string can indicate either that the input buffer is empty or that the character is a zero character. To determine which, check bit 3 of ST after the GET# is executed. If the bit is set (has a value of 8), the input buffer is empty. If the bit is clear (value of 0), the character is a CHR$(0).

IF ST AND8 THEN input buffer is empty
IF (ST AND8) = 0 THEN it's a CHR$(0)

The CHR$(0) normally is not used in ASCII communication and can just be ignored. It's important, though, in such activities as transferring a program file by modem, in which case every character is important.

Summary

- RS-232, like ASCII, is one of the few standards in the computer industry. It specifies how computers and devices communicate.
- The RS-232 convention applies to serial communication, where bits of a byte are sent one at a time. Parallel communication, the other method of data transmission, transfers all eight bits at the same time.
- The modem is used to convert bits into tones that can be carried by telephone lines to another computer which converts the tones back into bits. This process is called modulation and demodulation.
- The modem can transmit and receive at the same time. No direction has to be specified as part of the OPEN statement.

- A buffer is a temporary character-holding place. An input buffer insures that no incoming characters will be lost if some come in too fast for the computer to process. An output buffer holds characters until the modem is ready to transmit them.
- The syntax for OPEN to open an RS-232 file is the standard OPEN *file number, device number, secondary address, string.*
- File numbers from 1 to 127 are most frequently used. File numbers from 128 to 255 can be used with a serial printer which needs linefeed characters. The device number for an RS-232 device is 2. The secondary address is not used and should be set to 0. The string is one or two characters. The first character is a control character which sets the number of stop bits, the word length, and the baud rate. The second character is optional and is rarely used. It sets the parity, duplex, and handshaking.
- When an RS-232 OPEN is executed, 512 bytes are taken from free memory and are used for buffers. If there are not 512 bytes available, the end of the program is overwritten.
- The OPEN performs a CLR, which erases all variables, arrays, and defined functions, and closes all open files. An OPEN to an RS-232 device should be done as part of the program's initialization, before the assignments, dimensions, definitions, and OPENs. A CLR is also performed when a file to an RS-232 device is closed.
- Only one file to an RS-232 device can be open at any time. Communication with other devices is not allowed when an RS-232 file is open.
- With RS-232 communication, the status variable, ST, indicates errors.
- Translation is required to convert between ASCII and Commodore 64 character codes.