EXATRON STRINGY FLOPPY

USERS MANUAL

TRS—80 Model I
and
TRS—80 Model III

June 1982

Written by Bill Burnham

Edited by Jim Perry

exatron

excellence in electronics

Exatron Corporation, 181 Commercial St., Sunnyvale, CA 94086

EXATRON STRINGY FLOPPY

USERS MANUAL

TRS—80 Model I
and
TRS—80 Model III

June 1982

Written by Bill Burnham

Edited by Jim Perry

exatron

excellence in electronics

Exatron Corporation, 181 Commercial St., Sunnyvale, CA 94086

CONTENTS ==

Special thanks to: Tom Warfield, Jack Smock, Dana Huff, Gerald Melin, Brad Kidder, Richard Bidleman and I. Webb for their review and comments on the drafts of this manual.

INTRODUCTION ===

Congratulations on being the owner of an Exatron Stringy Floppy (ESF), the low—cost mass—storage alternative! Nearly all new owners “fall in love” with their Stringy Floppies. We hope you do also.

Use this manual to learn all you can about your Stringy Floppy, we want you to get the most out of it. The 30-day unconditional money—back guarantee is one of Exatron ‘s ways of making certain that all ESF users are satisfied users. If, for any reason, you find that the ESF does not live up to your expectations, then take advantage of the guarantee.

On the other hand, if you are satisfied, spread the word by telling other TRS-80 owners about it - or better yet, show them how it works. Take it to a local users group meeting for a demonstration, or even start an ESF Owners Association Workshop.

Your comments and suggestions, both about the ESF itself and this manual, are welcome. We hope never to get too big to listen to our customers. Call on the toll—free Hot Line:

(800) 538 8559 (inside California 408 737—7111)
9.00am - 5.00 pm PST

WHAT IS A STRINGY FLOPPY? ==

The Exatron Stringy Floppy (ESF) is a unique low-cost, high-speed, extremely—compact and reliable data—storage system for use with microcomputer systems that fits the gap, between cassettes and disk drives, perfectly.

The system is based on a miniature endless—loop tape cartridge (called a “wafer”), a precision direct—drive transport mechanism, and associated electronic circuitry.

Each wafer, smaller than most business cards, is only 2—3/4” long by 1—9/16” wide and 3/16” thick. The wafer contains an endless loop of magnetic tape (from 5— to 50—feet long), that is 1/16” wide. The amount of data that can be recorded on a wafer depends on the length of tape in it, typically, a 50—foot wafer can hold at least 40—thousand bytes of data.

With only one moving part (the motor spindle), the transport mechanism is extremely reliable. The motor spindle is used as a “direct drive” on the tape, pulling it past the record/replay head at a constant speed of 10 inches per second. With a 5—foot wafer a complete tape cycle takes less than 6 seconds.

The recording technique used in the ESF system is called “bi—phase recording”, which is the same as that used in disk systems. This special electronic technique results in phenomenal reliability, even when the motor speed varies by as much as ten percent. The entire system continues to function under extremely adverse conditions.

Solid—state optical sensors are used to detect the physical presence and write—protection status of the wafer. A reflective splice in the tape loop, which acts as an end—of—tape/beginning—of—tape mark, is also detected optically.

IMPORTANT READ THIS PAGE! ==

MODEL I SYSTEM REQUIREMENTS ==

The Model I ESF requires Level II BASIC and a minimum 16K of memory in your Model I TRS—80.

It will work with any other configuration of the Model I.

The ESF operating system is in a 2K Read Only Memory inside the ESF unit.

MODEL III SYSTEM REQUIREMENTS ==

The ESF III will only work with a Model III that has 48K of RAM installed. If your computer has 16K or 32K only, then Exatron can supply additional RAM for self installation (see the current Price Sheet for cost).

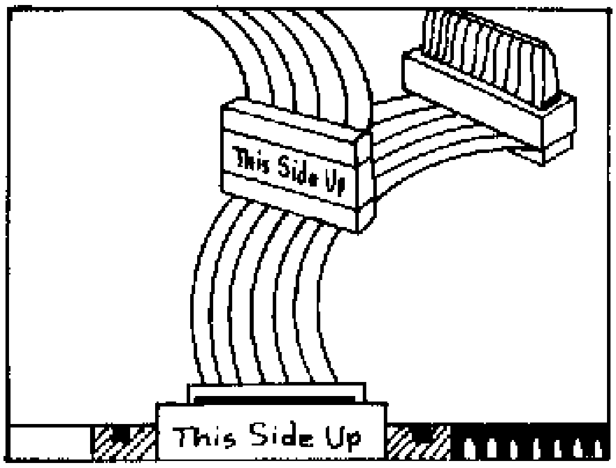
The ESF III operating system software is loads into the top 2K of your memory (4K with Data I/O). This will conflict with some commercial machine-language software, if it loads into the top of memory also.

At present Exatron has no plans for a relocatable version of the ES? III software.

OPERATION WITH DISK DRIVES

Both the Model I and Model III units will operate on a system with disk drives.

When in Disk BASIC issue the command CMD”T”, to disable the interrupts, then initialize the ESF in the normal way. Use the SYSTEM response /12346 with the Model I ESF, and ESF for the Model III.


This Side Up
This Side Up

exatron

MODEL I INSTALLATION ==

First make sure that your Model I and any connected peripherals are working properly. If all seems to be normal then turn everything off and unplug the power.

Inspect the ESF and the two—for-one bus—extender cable for any obvious damage. Pay particular attention to the contacts at the end of the 40-way ribbon cable connected to the ESF and on the bus-extender cable. These contacts should all be straight and even. If you see any bent contacts read the information on the next page.

Carefully remove any peripherals that you may have connected to the expansion—port card edge of the keyboard: e.g., expansion interface, printer, etc. and connect the female connector of the bus—extender cable to the keyboard card edge. Make certain that the “THIS SIDE UP” label is readable from the front of the keyboard.

Exercise extreme caution in making this connection in order not to bend any of the contact fingers in the bus—extender connector. Use a slow, end—to—end, rocking motion, with constant pressure toward the card edge. Do not use undue force. If the connector is a very tight fit read the information on the next page.

After connecting the bus—extender cable to the expansion—port card edge, plug the ESF ribbon—cable connector into the bus—extender in the position nearest to the keyboard. Again, take care in making the connection and check that the “THIS SIDE UP” label on the ESF ribbon—cable connector can be read from the front of the keyboard.

Now connect any device that you may have had connected to the expansion—port card edge to the remaining male connector at the end of the bus—extender cable.

For most reliable performance, your ESF should always be connected to the bus—extender cable, not directly to the expansion port, and should be the device connected nearest to the card—edge connector.

If you add more ESF drives to your system (you may have up to eight ESF units connected at one time) you can order a bus—extender cable with multiple connectors. These special Exatron cables can also be used with any other peripherals that need to be connected to the keyboard.

THICK CARD EDGES ==

If you feel a lot of resistance when plugging in the bus extender the probable cause is a slightly thick expansion-port card edge. In this case you can bevel the top and bottom of the expansion–port card edge, very slightly, with a fine file or emery board.

Take care not to get any filings inside the computer, and draw the file only away from the card edge — not toward it nor sideways —so as not to damage the metal fingers on the card edge.

After the beveling operation, brush away any debris and clean the contact fingers by lightly rubbing them with a soft eraser. The bus–extender connector should now fit easily and snugly over the card edge.

FIXING BENT CONNECTOR CONTACTS ======================================

If the contact fingers appear to be bent then you can straighten them with a small pointed plastic or wooden stick, such as a round toothpick. Usually, though, all that is required to correct any connector problems is simply to break and remake the connection. Some oxidation may have contaminated the contact surfaces, and the wiping action of breaking and remaking the connection is enough to clean the contacts and ensure a trouble–free connection between the two connectors.

POWER TO THE EFS ==

Connected to the ESF unit, by a two–wire cord, is a small plug-in transformer that converts the 117–volt AC household voltage to the correct AC voltage for the ESF. As the ESF does not have a power On/Off switch, we recommend that the transformer is connected so that a master switch turns it off when you switch off the computer.

No harm will come to the ESF or transformer if the power is left switched on when not in use, but, it will consume a small amount of energy — like an electric clock does.

MAKING IT WORK - MODEL I ==

First, restore power to your computer system (which now, of course, includes the ESF) and check that everything in your system still works correctly.

Now you need to let the computer know that a Stringy Floppy has been attached. Switch off the computer for 10 seconds, then switch on and answer the MEMORY SIZE? question by pressing Enter.

Later, you can answer the MEMORY SIZE? with whatever value you would normally use, but for this get—acquainted session just let the system default to its maximum available memory.

When the READY prompt appears after you answer “MEMORY SIZE?”, type :

```
SYSTEM (Enter)
```

A new prompt should appear

```
*?_
```

Answer it by typing:

```
/12345 (Enter)
```

You should now see the ESF sign—on message appear on the screen. The message should read:

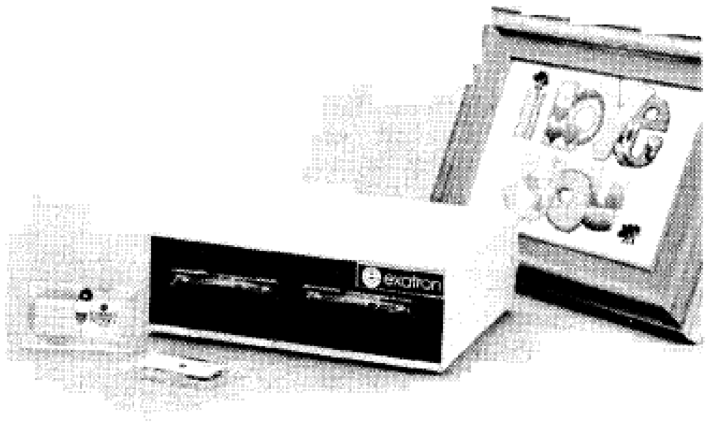
```
EXATRON STRINGY FLOPPY VERSION 4.1

>_
```

If you do not get this sign—on message and the computer hangs up, probably one of two things has happened.

It could be that you did not answer the *? prompt correctly. (It’s easy to forget to type the slash before typing the numbers.) Try pressing the RESET button, and start again. If you are sure that you‘ve answered the *? prompt correctly, then the next most likely cause is a bad connection at the point where the ribbon cable from the ES? connects to the bus—extender cable.

In this case turn off the power and check the connection, paying particular attention to the female connector on the EFS ribbon cable. Sometimes the contact fingers can get misaligned and not mate properly with the male bus—extender connector. This is not a common occurrence, but it can happen. Also, make certain that the connectors are the correct way up.

In the unlikely event that you still cannot get the sign-on message, after investigating these probable causes, then please call Exatron on the Hot Line and get advice from the Service Department.


exatron

MODEL III INSTALLATION ==

At present the ESF III will only work with a 48K Model III — if your machine has less memory read page 6, before proceeding.

First make sure that your Model III and any connected peripherals are working properly. If all seems to be normal then turn everything off and unplug the power.

Inspect the ESF for any obvious damage. Pay particular attention to the contacts at the end of the 50-way ribbon cable connected to the ESF. These contacts should all be straight and even. If you see anything suspicious, call Exatron and get advice from our Service Department.

Carefully remove any peripherals that you may have connected to the expansion—port card edge (in the center at the back of the Model III).

Connect the 50—way ESF connector to the expansion port, so that the cable exits from the rear of the computer. Exercise extreme caution in making this connection in order to not bend any of the contact fingers in the connector. Use a slow, end—to—end, rocking motion, with constant pressure toward the card edge. Do not use undue force.

IMPORTANT: DON’T PLACE YOUR ESF III ON TOP OF THE COMPUTER – PUT IT ON THE TABLE TO THE LEFT OR RIGHT OF THE COMPUTER.

Now connect the 5—way DIN plug of the Radio Shack cassette cable to the Model III cassette port. The three miniature jacks, at the other end of the cable, are inserted into the rear of the ESF as follows:

1. The large BLACK jack to the socket nearest the 50—way cable.
2. The large GRAY jack to the socket furthest from 50—way cable.
3. The remaining SMALL GRAY jack to the middle socket.

Read the section on page 9 about power, then plug the ESF transformer in. This completes the physical installation of the ESF III, and you are now ready to check out your complete system.

MAKING IT WORK - MODEL III ==

First, restore power to your Model III (which now, of course, has a Stringy Floppy connected) and check that everything still works correctly.

Now you need to let the computer know that a Stringy Floppy has been attached. Switch off the computer for 10 seconds, then switch on and answer both the CASS? and MEMORY SIZE? questions by pressing Enter.

At this point insert the wafer marked “MOD III BOOT WAFER” into the ESF drive, (if you have a dual—drive version use the lefthand drive.) As you push it in you should hear a high-pitched “beep” from the ESF.

When the READY prompt appears after you answer “MEMORY SIZE?”, type:

```
SYSTEM (Enter)
```

A new prompt should appear

```
*?_
```

Answer it by typing:

```
ESF (Enter)
```

The right—hand red light on the ESF will come on, and you should hear a buzz from the ESF motor. Within 10 seconds you should hear a “beep”, like when you inserted the wafer, and the motor buzz will drop in pitch.

In the top right of the display twin asterisks will flash, for a few seconds, followed by:

```
ESF Data I/O Version 4.la
Exatron Stringy Floppy Version 4.1
Reading..
```

This should be followed by;

```
      Done

ESF Bootwriter

This program can be used to:
(1) Copy itself ——— ESF Bootwriter
(2) Make a custom ESF Boot wafer
(3) Just bootstrap the ESF software
Which function do you want (1, 2 or 3) ?
```

After you‘ve loaded the “Boot” wafer, you need to make some copies of it. To do this respond 1 to the display prompt, and then press the enter key. You will be asked to remove the original boot wafer, and insert a blank 10–ft wafer:

```
Please take out the ESF Bootwriter wafer, and
insert a 10 ft blank wafer in Drive 0.
(it must be write-enabled)
Press (ENTER) key to start ? _
```

After you insert a new wafer, and press enter, the program will make a new boot wafer automatically. Then it will ask you to test the wafer, and if you want to make another copy:

```
Writing. . Done
Writing. . Done
Please write-protect the wafer and test it
Do you want to write another one (Y or N) ?
```

Before you test the wafer, make another copy by inserting another blank 10-ft wafer and replying “Y”. Then test both your new boot wafers by switching off and following the startup sequence with each in turn.

After you have a couple of extra boot wafers you can make a custom boot wafer, without the Data I/O or Bootwriter programs if you wish. The bootwriter program options 2 and 3 prompt you in a similar way to the number 1 option, just do what the messages tell you to do. With a custom Boot wafer you just have to type ESF, in response to the SYSTEM prompt to load the ESF software.

WHAT IS A “BOOT” PROGRAM ? ==

When you first connect the ESF III to your Model III it fools the computer into thinking it is a cassette deck. The “BOOT” wafer contains a special program that patches into the BASIC interpreter, and loads the ESF operating system into the top of memory. After this the ESF does not need to impersonate a cassette deck.

The term “BOOT” signifies a program that is designed to help load another program -- like lifting yourself up by your own bootstraps!

MORE ABOUT THE ESF III SOFTWARE =====================================

In the original Model I TRS—80 there was an unused 2K-byte section of memory, which is where the Model I ESF ROM is addressed, when Radio Shack produced the Model III they used up this part of memory with enhancements to the Microsoft BASIC.

So, for the ESF III we had to move the Stringy Floppy software into the top of RAM (booting it in with the aid of the cassette port). All the machine-language subroutines used by the Model I and III ESF are in the same relative locations —— for more details send for a copy of the Advanced Programmers Guide, which contains the complete source code.

IF NOTHING WORKS ==

If you do not get these sign—on messages, or the computer hangs up, or you hear several “beeps” then one of two things has probably happened.

It could be that you did not answer the *? prompt correctly. Try pressing the RESET button, and start again.

If you are sure that you’ve answered the *? prompt correctly, then the next most likely cause is a bad connection at the point where the ribbon cable from the ESF connects to the Model III.

In this case turn off the power and check the female connector on the ESF ribbon cable. Sometimes the contact fingers can get misalign and do not mate properly with the card edge. This is not a common occurrence, but it can happen. Also, make certain that the cassette lead is connected correctly.

In the unlikely event that you still cannot get the sign—on message, after investigating these probable causes, then there may be a fault within the ESF, the “Boot” wafer, or the connecting cable itself. If this appears to be the case, then please call Exatron on the Hot Line and get advice from the Service Department.

THE ESF COMMANDS==

All ESF commands are preceded by the “@“ symbol and may be used as statements within a BASIC program or entered as commands while the computer is in the command mode. This section covers the use of these commands while in the command mode. Their use as statements within a BASIC program will be covered in a later section.

@NEW ===

@NEW is the ESF electronic equivalent of an eraser, it will erase any data that was previously recorded, so think twice before using this command. Normally, the @NEW command is used when you want to get rid of all data from an old wafer or certify a new wafer.

The @NEW command may be entered with or without a file number as a suffix. But note that @NEW, @NEW 0 and @NEW 1 will all cause the entire wafer to be overwritten.

Entering the command @NEW with a file number greater than 1, but less than 99, will cause only that file and any higher-numbered files to be written over.

After execution the @NEW command will display the number of bytes available on the wafer.

The complete message that will appear on the screen at the end of the @NEW (certify) operation would look similar to the following:

```
>@NEW
ERASING. .4024 BYTES. .DONE READY
>_
```

The byte-count value depends upon the length of tape in the wafer.

Files with numbers below the suffix given to the @NEW command are not overwritten.

If you have a wafer with five files recorded on it, and you enter the command “@NEW 3”, then files 3, 4, and 5 will be overwritten but files 1 and 2 will still be intact.

Do not give an @NEW file number more than one higher than the last recorded file number, as the ESF software will be looking for the end of the previous file to know when to start the certify operation. It will, of course, never find it, because it was not recorded. When this happens, the ESF will run continuously and you must press the BREAK key to stop the fruitless search.

HOW MANY BYTES ARE LEFT?===

The byte—count feature can be useful in determining how many bytes are left available on a tape. To determine this information, simply enter the command “@NEW” with a file number one higher than the last file number that you have recorded on the wafer.

For example, if you have five files recorded on the wafer, then enter the command “@NEW 6”, and the wafer will be certified from the end of file 5 to the end of tape and give you the corresponding bytes available for data storage.

HOW @NEW WORKS ==

When the ESF operating system receives the @NEW command, the following things happen:

The message “ERASING..” will appear on the display, the ESF tape—transport motor turns on, and the endless—loop wafer tape will be pulled past the read/write head of the ESF. An optical sensor within the ESF checks for the end-of-tape / beginning-of-tape (EOT/BOT) reflective splice.

During this period, the motor-on indicator (right—hand light on the front of the ESF unit) lights up. When the splice is detected, the write indicator (left-hand light on the front of the ESF unit) is switched on, marker data and a digital pattern is recorded (written) on the full length of the tape by the ESF software.

When the write operation has been completed, the write light will go out and the ESF will shift into a read operation. This is sometimes referred to as the “verify cycle” of the @NEW command.

The ESF continues to transport the tape across the head while reading (verifying) the information that was recorded during the just—completed write operation. It reads back the entire tape; and if the data that is read back is exactly the same as recorded, the tape transport motor and motor—on light are then turned off.

At this time, a byte—count message is displayed and the word “DONE” appears. The byte—count message is telling you how many bytes of data you can expect to record on that wafer with a future @SAVE command.

The data recorded at the beginning of this @NEW operation is recognized by the ESF software as a “FILE MARK 0” and indicates where to begin recording the file 1 data of a future @SAVE operation.

@SAVE ===

@SAVE is the command to copy a BASIC program from computer memory to wafer. A variation of this command is used to save a machine–language program and is covered in detail in its own section.

The @SAVE command must always have a file number suffix, which must be at least 1.

Attempting to @SAVE a program without a file number, or using a file number 0, will result in a syntax error.

All files must be saved in sequential order, starting with 1 and progressing up to a maximum of 99. There can be no gaps in the file numbers on a wafer.

If the program that you want to save will be the first program on a wafer, then the proper command is:

```
@SAVE 1 (ENTER)
```

This will result in the following display:

```
@SAVE 1
WRITING. .DONE

READY
>_
_
```

The next program to be saved on the same wafer must be file 2 followed by 3, and so on. As long as you continue to save multiple program files on the same wafer, you would continue saving programs in this sequential manner. The ESF software will allow up to 99 files to be saved on one wafer.

The actual number of program files that you can save is limited to a combination of wafer and file length.

HOW @SAVE WORKS ==

The observed operation of the ESF during an @SAVE operation is exactly the same as during an @NEW operation, except for the messages displayed on the screen. The sequence of action is as follows:

If this were the first program to be saved on the wafer, then an “@SAVE 1” command would be issued in the command mode of the computer. When the ESF software receives this command, the message “WRITING..” appears on the screen and the tape transport motor and its on/off indicator LED will light as in the @145W operation.

When the file 0 marker (recorded at the beginning of the tape by the @NEW operation) has been found, the ESF software will begin writing the program data to wafer. At this time, the write-indicator light will turn on and will remain on until all the program has been written to wafer.

The last operation to be performed by the ESF while in this write cycle is to record an “end of file 1” marker after the last byte of program data has been recorded. The ESF then shifts into its read (verify) cycle and continues just as in the verify cycle of the @NEW operation.

When the verify cycle has completed, the ESF motor will stop, the motor-on light will turn off, and the message “DONE” will appear on the screen. The complete message that should appear on the screen after a successful @SAVE operation is:

```
>@SAVE 1 WRITING. .DONE READY
>_
_
```

When the next program is to be saved, the operation is the same, except that the write cycle will not start until the “end of file” marker of the last saved program is detected. In this case, the next program should be saved with the command “@SAVE 2”, and the write cycle will start after the detection of the “end of file 1” marker.

It should be obvious here what would happen if you tried to @SAVE this program with a file number of 3 or higher. The ESF would run forever looking for that “end of file” marker that was never recorded. Again, as in the @NEW operation, you must press the BREAK key to stop the fruitless search.

@LOAD ===

The @LOAD command is used whenever you want to load a program from wafer into the computer. If you want to load a specific program file, then you must add a file number suffix.

If you simply enter the @LOAD command without a file number then the next file, that the ESF software finds, will be loaded.

If you issue an @LOAD command with a nonexistent file number (such as “@LOAD 6”, and there are only five files recorded on the wafer), then the ESF will run forever —— looking for a nonexistent file. The BREAK key must be pressed in order to break out of the fruitless search cycle.

The command “@LOAD 0” will give the same results as “@LOAD”. Of course, if there is only one program on the wafer, or if you are interested in only loading the next file, then a simple “@LOAD” command will suffice.

If a wafer has several files recorded on it and you want to load in file 3, you would enter the command as “@LOAD 3.” Of course, if you had previously loaded file 2, then again, a simple “@LOAD” without a file number would suffice.

HOW @LOAD WORKS ===

When the ESF software receives The @LOAD command, it causes the message “READING..” to appear on the screen. At the same time, the motor and its indicator light will turn on.

If the command issued were a simple “@LOAD” without a file number, then the ESF would begin reading in the program file immediately following the first “end of file” marker that it encountered. (The file 0 mark, at the beginning of the tape, is accepted by ESF as an “end of file” marker); however, if the @LOAD command were issued with a file number, such as “@LOAD 3”, the read operation would not begin until the “end of file 2” marker had been detected.

After a successful @LOAD operation has been completed, the message “DONE” will appear on the screen and the ESF motor and its indicator light will turn off. The completed message should read as follows:

```
>@LOAD 3
READING...DONE
>_
```

SYSTEM /12345 AND OTHER VARIATIONS ==================================

This might be an opportune time to discuss the meaning behind answering the SYSTEM prompt, *?, with the address number of /12345.

Actually, the ESF can be initialized by answering the SYSTEM prompt, *?, with one of seven different addresses. These are:

```
        /12340,  /12341, /12342,
        /12343,  /12344, /12345,   /12346
```

The reason for choosing 12345 as the example address will become apparent as we proceed through this discussion.

There is a 2-K area in the Level II ROM, beginning at address 12288, that is not used by the TRS-80. When you answer the SYSTEM prompt with any of the above seven addresses, you will have selected a valid entry point into the ESF ROM that will initiate a set of instructions to be executed in the ESF ROM routines. The exact instructions that will be executed depend on which entry address you chose in answer to the SYSTEM prompt.

No matter which entry—point address you choose, the following action is common to all:

The ESF ROM routines check for the top of available memory as set by your answer to the MEMORY SIZE? question at power up. The ESF firmware will then write four bytes of machine-language code at the top of available memory and then reset MEMORY SIZE to a value that is four less than what was selected at power up.

In addition to the action described above (which will occur on any of the seven entry—point addresses), when /12345 is selected as the entry—point address, a keyboard debounce routine is initiated within the ESF firmware and the appropriate sign-on message is displayed on the monitor screen. In this case, the sign—on message is

```
EXATRON STRINGY FLOPPY VERSION 4.1

>_
```

Answering the SYSTEM prompt with /12346 will give the same results as /12345, with the exception that no debounce routine will be initiated. This is useful when using disks, as all disk operating systems have special keyboard routines.

Choosing an entry—point address of 12340, 12341, 12342, 12343, or 12344 is really a variation of the @LOAD command.

Each of these will initialize the ESF ROM, with debounce, then load the file represented by the last digit of the 1234n number.

For example, /12340 will load the next file that is found; /12341 will load file 1; /12342 will load file 2.

PARITY, CHECKSUM AND VERIFY ERRORS====================================

All of the examples given so far have been for successful commands. So what happens if something goes wrong?

If, during the verify cycle of an @145W or an @SAVE command or during the read operation of an @LOAD command, an error is detected, the ESF stops and gives an error message. After the error message the ESF software returns you to BASIC. Error messages as they might appear on the monitor screen are as follows:

```
>@NEW
ERASING...254 BYTES...PARITY ERROR
?FD ERROR
READY
>_


>@SAVE 1 WRITING...VERIFY ERROR ?
FD ERROR
READY
>_


>@LOAD 1
READING...PARITY ERROR
?FD ERROR
READY
>_
```

You may occasionally see a “CHECKSUM ERROR” message rather than a “PARITY ERROR” message. The message depends on the type of error detected during the read operation.

We suggest that if you detect any of these errors while trying a command, you repeat the command a couple of times. If this does not work then you can try cleaning the read/write head – see the Appendix 1.

WRITE-PROTECTED ERROR ==

This error can happen during an @NEW or @SAVE, and means that you tried to write data to a wafer that has had the write-protect sticker removed. This is the small, white, round sticker that is attached just above the EOT/BOT sensor hole cut out on the label side of the wafer.

This sticker must be in place before any write operation can be performed. You would normally remove this sticker from any programmed wafer that you wanted to protect from accidental erasure. Replacement stickers can be purchased locally at most stationary stores.

TAPE TOO SHORT ERROR ==

If you try to @SAVE a program that is too long to fit in the space available on a wafer, then a "TAPE TOO SHORT" error message will be displayed, and you must use a longer wafer.

The screen display would appear as follows:

```
>@SAVE I
WRITING...TAPE TOO SHORT ERROR ?FD ERROR
READY
>_
```

IT WON'T STOP & OTHER ERRORS ==

One last error type that you may run across is failure to @SAVE a program file with a file number 1. This failure will not result in an error message but will be evidenced by the fact that the ESF never goes into its write made. It just keeps running until you press
BREAK.

The most probable cause for this type of error or malfunction is that the file 0 marker recorded during an @NEW operation has somehow been lost. This could be due to stray magnetic fields in an area where the wafer was kept. High ambient temperatures can also cause loss of data on magnetic tape. Therefore, always keep your wafers in an area free from magnetic fields and extremes in temperature.

Examples of bad places to leave your wafers would be on top of or next to the video monitor (strong electro—magnetic fields here) and on the dashboard of your car, where direct sunlight can wreak havoc with the magnetic media as well as the plastic wafer itself.

If you do find that you cannot perform an @SAVE 1 on a wafer, simply @NEWing that wafer again will usually solve the problem.

THE BREAK KEY IS BROKEN ===

The only tine that the BREAK key will not return you to Level II BASIC is when the ESF is in a write cycle, as evidenced by the write light being on. To break out of this cycle, you must press the RESET button. Whenever the BREAK key is pressed to brake out of an ESF searcn or read cycle, the error message "?FD ERROR" will appear on the screen. This is normal and can be disregarded.

EXERCISING THE STRINGY FLOPPY COMMANDS ==============================

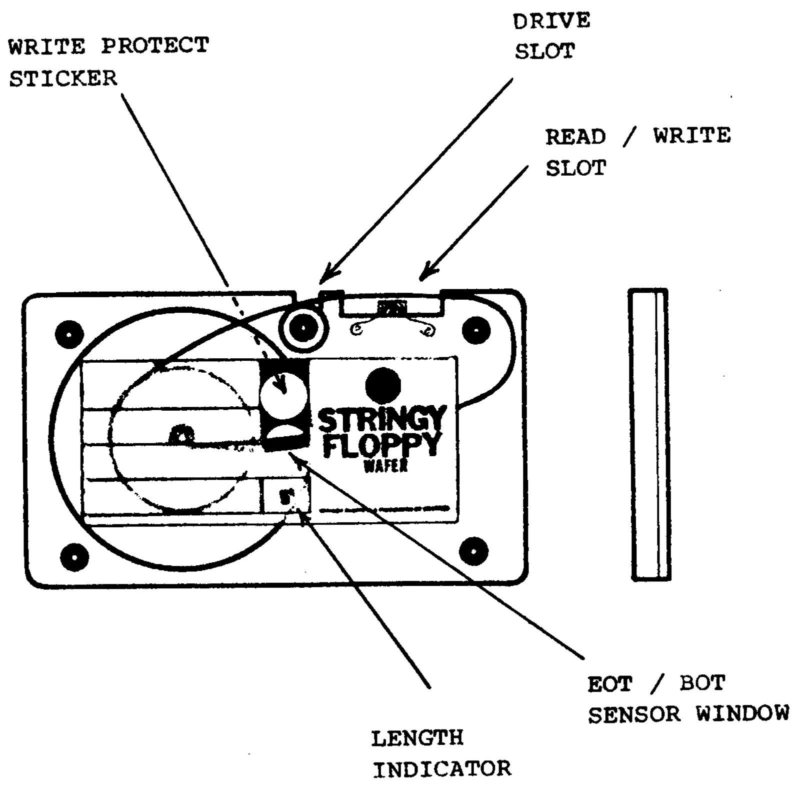
Now that you have the ESF connected to your computer system and have successfully got the sign on—message ...

EXATRON STRINGY FLOPPY VERSION 4.1

... you are ready to get some hands—on experience in exercising the three ESF commands now available.

Before the step—by-step examples for ESF commands, we will discuss the ESF wafer itself. After this, you will be ready to insert a wafer into the ESF unit and begin using the three ESF commands:
@NEW, @SAVE, and @LOAD.

MEET THE WAFER ==



END-AND-BEGINNING—OF-TAPE SENSOR ====================================

Just above the wafer—length indicator is a circular cutout in the wafer label. This is the window that an optical sensor in the ESF looks through to detect the reflective splice that joins the two ends of tape together, (how else did you think we made endless tape?) This splice is called “the end of tape / beginning of tape marker”
(EOT/BOT.)

This sensor window must never be covered. If it is, say by a wafer-identifying label that you might stick on the face of the wafer, an @NEW operation cannot be performed.

THE WRITE PROTECT STICKER ===

Above this sensor window is the area where the write-protect sticker is attached. This round, white sticker must be in place in order to write data to the tape.

Peel this sticker off when you want to protect the tape from being inadvertently overwritten. Make sure no reflective material is left in this area.

DRIVE AND READ/WRITE SLOTS ==

Just above and a little to the right of the write-protect sticker is a small cutout in the top of the plastic wafer. This is the slot where the ESF capstan spindle engages the tape and causes the tape to traverse from right to left across the ESF Read/Write head.

To the right of the spindle slot is another cutout in the top of the plastic wafer. This is the area where the ESF Read/Write head presses against the tape so that it can record and replay data.

ESF COMMAND EXERCISES ===

We will assume from this point on that your computer system is up and running with the ESF correctly initialized.

All exercises in this section will be done in the command mode of your computer. Using these commands as statements within a BASIC program will be covered in the next section.

Refer to page 19 for the meaning of any error messages that you may get while doing these exercises.

TRYING OUT @NEW ===

Okay, insert your new 5—ft blank wafer (label side up) into the mouth of the ESF. The design of the wafer is such that it can be inserted into the ESF only the correct way round (famous last words!)

As you insert the wafer, you will feel a slight resistance to further movement at about the point where the bottom of the water label is even with the lip of the ESF slot mouth. Push the wafer past this resistance point, and you will feel the wafer snap into place. You are now ready to issue ESF commands.

Type “@NEW”, and press Enter. The message “ERASING..” will appear on the screen, and the right-hand motor-on light of the ESF will turn on. When the EOT/BOT splice has been detected, the left—hand write light will turn on.

After the tape has been fully written, the write light will turn off and the data now written will be verified. After verification, the ESF unit and motor-on light will turn off and the byte—count message along with the word “DONE” will be displayed.

The number of bytes in the byte-count message is determined by the length of tape that is being certified. The byte count should not be less than the following for different length tapes:

5 ft at least 4,000 bytes	10 ft at least 8,000 bytes
20 ft at least 16,000 bytes	35 ft at least 28,000 bytes
50 ft at least 40,000 bytes	

NOTE: The ESF software can only count to 65,535, and then resets to zero, so if you get a low byte count with 50—foot wafers don ‘t panic

just add 65,535 to the displayed number!

Remove the 5—ft wafer, insert another new blank wafer of a different length, and repeat the @NEW command. You should notice the exact same sequence of events happening, with the exception that it takes longer and the displayed byte count is higher. It should take about 15 seconds to certify a 5-ft wafer, 30 seconds for a 10—ft wafer, and correspondingly longer periods for longer wafers.

TESTING BRAND-NEW WAFERS ==

We recommend that all new wafers be given a certifying test immediately upon receipt. This is best done by putting each of them through an @NEW operation five times in succession.

Although all wafers are tested before shipment, things can go wrong in transit, and this is a quick way to determine any hidden faults. If problems with the wafer or tape are discovered during this testing operation, the defective wafers should be returned to Exatron for replacement in accordance with the stated warranty.

If you encounter any errors during the @NEW operation, you should clean the read/write head again and repeat the test. Sometimes, new tape will have a tendency to leave some tape oxide contamination on the read/write head that will impair a good read/write operation.

Sometimes during an @NEW operation you may receive an error message and the wafer is not defective. This will be when the @NEW operation is terminated with an error message, but the indicated byte count is at least the amount stated in the beginning of this section.

This situation can occur sometimes just as the splice pulls out of the takeup hub within the wafer. There may be a momentary fluctuation in tape speed at this moment that could result in a PARITY ERROR being registered and the @NEW operation being prematurely terminated.

Since this type of error would be generated at the very end of the tape, the error can be disregarded and the wafer should be considered as having passed the test as long as the displayed byte count is not below the minimum listed for that length wafer.

The following is a suggested way to set up this test operation:

Insert a new, 5–ft. blank wafer and enter the following one line program:

```
10 FOR I = 1 TO 5: @NEW: NEXT I
```

Type “RUN” and press Enter. The @NEW operation will now repeat itself five times as the @NEW command is looped through the FOR/NEXT loop of the above program.

You may notice that the displayed byte count changes after each passage through the loop. This is normal and should be of no concern. It is caused by the fact that the tape passes by the read/write head a little faster, the longer the ESF is in operation.

The lubricant that is a part of the tape composition begins to work more effectively as the head and tape begin to warm due to surface friction. It is just during the first few @NEW cycles that this change in byte count (normally not more than a few hundred bytes)
It does not affect the operation of the ESF in any way.

TRYING OUT @SAVE ==

Insert a certified 5-ft wafer (the one that you just @NEWed) into the ESF and key in (or CLOAD from cassette) a small BASIC program. A program in the neighborhood of 500 bytes will be just fine. A program within this range of bytes will allow the completion of the following series of experiments using the “@SAVE” command. Add a line at the beginning ~f your program that says:

```
1 REM THIS IS PROGRAM #1
```

Type “@SAVE 1” and press Enter.

You will note that the same cycle of events will occur as in the @NEW operation, except that the displayed messages will be different. In this case, you will see the message “WRITING..”, followed by the word “DONE” when the operation is complete.

Now change line 1 to:

```
1 REM THIS IS PROGRAM #2
```

Save the program on the same wafer by typing “@SAVE 2”, and press Enter. You should observe the same action as before and now have two program files on your wafer: file 1 and file 2.

Save the same program again on the wafer, but this time do it with an “@SAVE 4” command. You will notice that the ESF never goes into its write cycle, because it cannot find an “end of file 3” marker.

Press the BREAK key to return to BASIC.

Try saving the same program without giving a file number to the @SAVE command. Type “@SAVE”, and press Enter. Notice that the computer will not accept this command and returns a syntax error. All @SAVE commands must have a file—designator number.

Edit the first program line again and make it read:

```
1 REM THIS IS PROGRAM #3
```

Save the program on the same wafer as file 3. Type “@SAVE 3” and press Enter.

TRYING OUT @LOAD ==

Type “NEW” (caution -- not “@NEW”) and press Enter. This will, in effect, clear out the resident program in your computer memory; and if you type “LIST” and press Enter, you will see only the READY prompt.

With the same wafer still in the ESF, issue an @LOAD command without a file number, type:

@LOAD (Enter)

You will notice that the message “READING..” will appear on the screen and that the ESF and its motor-on light will both turn on. After the program has loaded into memory, the word “DONE” will appear on the screen and the ESF and motor—on light will turn off.

Type “LIST” and press Enter. You will see that program number 1 has been loaded into memory. This shows that executing an @LOAD command without a file number will simply load in the next file on the wafer from wherever the tape may be resting at that point in time.

Type “@LOAD 3” and press Enter.

When the message “DONE” has been displayed, type “LIST” and press Enter. You should now see that program number 3 has overlayed program number 1 in memory. This action points out how, by giving a file number to an @LOAD command, one may cause the ESF to skip over any intervening files and load only the file given in the @LOAD command.

Type “@LOAD 4” and press Enter.

Notice how the message “READING..” is displayed, but the ESF never stops its search operation. This is, of course, because there never was a file 4 recorded on the wafer.

Press the BREAK key to terminate the search operation.

Type “@NEW 4” and press Enter. The ESF will go into a certifying operation and give a byte count equal to the number of bytes still available on the wafer.

Type “@LOAD” along with any file number you like from 1 to 3, and press Enter. After the message “DONE” has been displayed, type “LIST” and press Enter. You will see that you have not destroyed any files on the wafer by entering the command “@NEW 4.” -

Type “@NEW 5” and press Enter. Notice that the ESF never enters the write phase of the @NEW operation, because there is no “end of file 4” marker to initiate the writing of the certify data. Press BREAK to terminate the search operation.

List the program in memory and increase its byte length by adding several more lines of code. Type “@SAVE 2” and press Enter.

After the @SAVE operation has been completed, type “@LOAD 2” and press Enter. After the @LOAD operation has been completed, list the program to verify that the new program has indeed been aaved as a new file number 2.

Type “@LOAD 3” and press Enter. Notice that the ESF never completes the @LOAD operation. The new, longer file number 2 that you have just written has overlaid the beginning of file number 3.

The ESF software cannot find the beginning-of-program marker that is recorded with every @SAVE operation along with the “end of file” marker. You have lost file number 3. Press BREAK to terminate the search operation. If, by some chance, file 3 does load, then add some more lines of code to file 2 and try again.

The point here is that it is foolish to resave a file on a wafer that has file numbers higher than the one that you will be @SAVING. If the new file is longer than the old file of the same number, you definitely will lose access to the following file. If it is the same or slightly less in byte length, then possibly you may be able to pull it off; hut more than likely, the beginning file mark of the following program will still get clobbered because of slight variations in ESF motor/tape transport speed.

The only time that you can perform this operation and not run into trouble along these lines is when you are resaving the last file on the wafer. Here, the only limitation is that there be enough tape left on the wafer to hold the number of bytes required to @SAVE the file.

Type “@SAVE 3” and press Enter. When the write operation has been completed, type “@SAVE 4” and press Enter. Keep repeating this operation, increasing the @SAVE file number by one each time, until you receive a “TAPE TOO SHORT” message on the screen.

Type “@LOAD” with any file number up to, but not including, that last file number. This will show that the “TAPE TOO SHORT” message and its subsequent action of returning you to BASIC did not destroy any previous files. Of course, if you tried to @LOAD that last file, you would get an error message, because it is an incomplete file. Try it.

Now type “@NEW” and press Enter. When the certify operation has been completed, type “@LOAD” and press Enter. You will notice that the read operation never completes, and the ESF runs continuously. This shows that the @NEW command has effectively removed all traces of any previously recorded files. Press BREAK to terminate the search.

MAKING BACKUP COPIES OF SOFTWARE ====================================

Now that you are familiar with the use of the ESF commands, it is time to make a backup copy of the programs contained on the Starter Kit program wafer.

As with any software, whether it be recorded on ESF wafer, cassette, or disk, a backup copy should be immediately made and the original should be put away in a safe, secure place.

There are several programs recorded on this wafer. Some are BASIC programs and some are machine-language programs. Consult the contents page included with the Starter Kit to find out which programs are in BASIC and what their associated byte counts are.

We will not, at this time, make backup copies of the machine—language programs contained on that wafer. We will, instead, confine this section to making backup copies of BASIC programs and defer the procedure for making copies of machine—language programs until later.

Follow the step—by-step procedure below for each BASIC program on the Starter Kit program wafer.

1: Choose a new wafer that has sufficient length to hold the complete program that you want to save.

For example, if one of the programs that you want to save has an indicated byte-count length of 8K, then you should choose at least a 10-ft wafer, as a 5—ft wafer does not contain enough tape to hold it.

2: Certify the wafer with the @NEW command. Of course, if you already know that the chosen wafer is certified, you may skip this step.

3: Insert the Starter Kit program wafer, type

@LOAD (desired file number) (Enter)

4: When the BASIC program file has loaded, remove the Starter Kit program wafer and insert your certified wafer.

Type “@SAVE 1” and press Enter.

5: After the word “DONE” appears on the screen, remove the wafer from the ESF unit and peel off the write —protect sticker. You now have a verified backup copy of the BASIC program that you loaded.

6: Put the back—up wafer back into the ESF and type:

@LOAD (Enter)
The program on the wafer will now load into the computer. Check by running the program

If you want to, try “@NEW”, and press Enter. You will see the “WRITE-PROTECTED ERROR” message appear on the screen, and you will be returned to BASIC.

This shows that removing the WRITE-PROTECT sticker did, in fact, prevent you from overwriting the data on that wafer. (If by some chance the ESF writes all over the wafer - sorry about that, your ESF is broken! Call Exatron on the Hot Line.)

Repeat steps 1 through 6 for each of the BASIC programs on the Starter Kit program wafer.

This procedure will put a copy of each BASIC program on an individual wafer by itself. If you wanted instead to save more than one program on the same wafer as separate files, then you should choose a wafer with a length long enough to hold the combined files and save them in the order that you want them on the wafer.

Remember, all files must be @SAVED with file numbers in ascending sequential order.

SAVING AND LOADING MACHINE LANGUAGE ================================

To save a machine—language program, the ESF software must be directed to the area in memory where the machine-language program is. Then it must write the particular memory area out to wafer as a program file. To do this the ESF software must be given two parameters: the starting address in memory, and its length.

This is done by adding the information, in decimal, to the @SAVE command in the following way:

```
@SAVE 1, Start Address, Length
```

Notice that for correct syntax of this machine—language save command, a comma must separate the file number from the starting address and a comma must separate the starting address from the program length.

When you want to load your machine-language program back into the computer, it is done in exactly the same manner as is a BASIC program: i.e., through the use of the @LOAD command, with or without a file number. The ESF software, when it receives the @LOAD command, will load the file into the proper area of memory in accordance with the code that was written as part of the file header information when it. was @SAVED.

Since issuing an @SAVE command along with a starting address and byte length results in a memory dump of that designated area of memory, you can see that what is saved to wafer could be data that represents code other than a machine—language program. Therefore, using this format, you can save any area of computer memory for whatever purpose you like.

AUTOSTART ==

If you want the program to come up running (automatically execute upon being loaded), then the autostart address can be added to the end of the command, thus:

```
@SAVE 1, start Address, Length, Autostart Address
```

If you do not want your program to execute automatically (autostart’, then do not include an autostart address as part of the command.

Notice that the length figure and the autostart address must also he separated by a comma. The autostart address is an optional part of the command.

HIGH-MEMORY ADDRESSES ===

The format shown above has to be altered for machine–language programs residing above memory address 32767, the reason being that the ROM routine that processes this address information can handle integer values only in the range of –32768 to +32767.

Therefore, address information above 32767 must be processed as a negative number. This is most easily accomplished by subtracting 65536 from any address above 32767. Using this procedure, the @SAVE command for a machine–language program that has a starting address of 60000 and an autostart address of 60050 would be issued in the following format:

```
@SAVE 1, 60000–65536, 1000, 60050–65536
```

STOPPING AN AUTOSTART ===

To make a valid save of the original code of a machine–language program, you must do it before the program has executed; and of course you must know the saving parameters. The ESF software has a feature that will solve both of these problems at once. It is a function called “autostart override” and is brought into play by holding down the SHIFT key while loading in a machine-language program from wafer. Here is how it works:

When a machine–language program has loaded into memory, the ESF software checks to see if the SHIFT key is down. If it is, the software will prevent further action by the program and instead will display the starting address, the program length in bytes, and the autostart address in decimal. After the program parameters have been displayed, control will shift back to BASIC.

If after making your backup copy you want to run the program, then type “SYSTEM” and press Enter. Answer the *? prompt by typing “/“ and the autostart address, and then press Enter. The program will now execute.

A point of information here: If you note that the autostart address is displayed as 12309, then it means that the original program was not saved with an autostart address. In this case, the program documentation would have given directions as to how to execute the program, once it was loaded.

STARTER KIT BACKUP COPIES ==

Now let`s put this information to practical use by making backup copies of the machine-language programs on the Starter Kit programmed wafer. We will do this in the same manner that we made backup copies of the BASIC programs. Carry out the following steps:

1. Refer to the contents page of your Starter Kit, and determine which files on the Programmed Wafer are machine-language programs.

2. Choose a program for saving to a backup wafer, and select a blank certified wafer of the appropriate length. (Choose the ESF-80 Monitor program as the first one to save, so that it can be used as the example in this series of steps.)

3. Using the appropriate file number for the ESF-80 Monitor program, type "@LOAD (file number)" and press Enter.

4. Hold the SHIFT key down until the program has loaded and you see the save parameters displayed. After the program haS loaded, the screen display should appear as follows:

```
>@LOAD (file number) READING...DONE BREAK
17152, 2711, 17152 ?FD ERROR READY
```

The "BREAK" and "?FD ERROR" part of the display are of no consequence and can be disregarded. The pertinent information in the above display are the decimal numbers 17152 (start address), 2711 (byte length of the program), and 17152 (autostart address of the program).

5. Remove the Starter Kit wafer and insert your blank, certified wafer.

6. Type "@SAVE 1, 17152, 2711, 17152", and press Enter. The ESF will complete a write and verify cycle just as it did in the BASIC program save operation. Any error messages displayed during this operation are to be handled `in the same way as are for a BASIC program.

7. When the machine-language program has been successfully saved, you should remove the write-protect sticker and mark the wafer in some way indicating which program is on that wafer.

8. Repeat the above steps for the rest of the machine-language programs on the Starter Kit wafer.

You now have a backup copy of every program on the Starter Kit wafer. From this point on, you should use only these backup copies and save your original Starter Kit wafer for emergency use.

CASSETTE-BASED MACHINE-LANGUAGE PROGRAMS ===========================

For the purposes of the following discussion assume that the cassette-based machine-language program is a standard SYSTEM-formatted tape with no copy-protect measures imbedded in its code and that it was not saved with autostart.

Transferring to wafer when the save parameters are known

1. Insert a blank, certified wafer into the ESF and insert the cassette into your tape recorder.

2. Prepare your tape recorder for a read operation and SYSTEM load your program as you normally do.

3. When the program has loaded, answer the *? prompt with a /6681. This will return you to BASIC without executing the program.

4. Type and Enter the machine-language @SAVE command with the appropriate arguments.

Transferring to wafer when the save parameters are not known is a bit more complicated. Included on the Starter Kit program wafer is a program titled ESF-80 Monitor. It has a useful function that will call in a machine-language program from cassette, display the necessary addresses for saving to wafer, and then, allow you to @SAVE it to wafer - even if the program on cassette were saved with autostart.

The above transfer operation is accomplished through the "L" command of the ESF-80 Monitor. The steps necessary to perform this transfer are well documented in the ESF-80 Monitor user `s manual that is included with your ESF Starter Kit. Be sure to read the section on using the "L" command thoroughly before trying to make a transfer from cassette to wafer.

You will find that all numeric entries called for when using the Monitor must be in hexadecimal. Decimal entries are not allowed and will result in misleading and faulty information from the various functions made available by the Monitor. This seems to be the most common mistake that first-time users of the Monitor program make.

IT'S NOT STANDARD

The overriding criterion for a successful transfer operation is that the program must have been saved in the normal, straightforward SYSTEM-type format. If the program is in any other format, you cannot save a copy of it to wafer using the facilities of the Monitor program.

This will be the single most important reason for any problems encountered in attempting to transfer commercial cassette-based software to wafer. The only answer to this dilemma is in going through the machine code used in the program and determining what changes must be made in order to make a copy to wafer. This, of course, is no mean task and generally requires the services of an experienced programmer, well versed in the art of machine-language programming.

This is an example of a type of problem that an ESF Workshop can take on and usually solve. There is always a myriad of computer talent attending these Workshops, and this is one more important reason why every ESF owner should join and support his local Workshop. If you have solved any of these sticky transfer problems and would like to share your findings with other ESF owners, Exatron would be most appreciative of receiving your revelations and will make them available, with credit to you, to other ESF owners who may be experiencing troubles with the same program.

Some day, of course, all authors will be coding for the Stringy Floppy and there won `t be problems of this nature anymore. You can be of help in this regard if every time that you get a program that is not ESF compatible, you write to the author or software house and inquire about an ESF version being made available. Sooner or later, as the list of ESF owners grows and grows, they will get the message to provide more software that is written for the ESF as well as for cassette and disk.

ESF COMMANDS INSIDE BASIC PROGRAMS =================================

The three ESF commands, @NEW, @SAVE, and @LOAD, may be issued and executed as statements within a BASIC program (computer in execute mode) or as direct keyboard input while the computer is in the command mode. The syntax of the commands and the results of their execution are the same in either computer mode. There is an added benefit to using the @LOAD command in the execute mode in that it will allow program chaining. This technique is described at the end of this section.

The only restriction in the use of ESF commands in the execute mode (when a program is actually running), is that they cannot be used in any IF .... THEN statements; a syntax error will be returned if they are used in this way.

The commands must be used as solitary statements on separate program lines. They may, however, be used in a multiple-statement line, as long as the command is separated from any other part of the line by colons. Following are two examples of the correct use of ESF commands in programs:

```
      40    ON X GOTO 70, 90, 110
      50    .........
      60     .........
      70    @LOAD 2
      80    .........
      90    @SAVE 2
      100   .........
      110   @NEW  2
      120   .........
```

```
10 FOR I = 1 TO 5:
       @NEW:
       NEXT I
```

The following is an example of improper use of the ESF commands inside a program:

```
10 IF X = 5 THEN @LOAD 2 ELSE @NEW 2
```

At the end of this section there is a sample program, for you to key in and run, that demonstrates the use of ESF commands as statements within a BASIC program.

PROGRAM CHAINING ===

Program chaining is the process whereby a program statement, when executed, will load another program into computer memory and still retain any variables that were established before the load command was issued. These retained variables can be used by the new program as well as variables established by the new program in the normal way.

If you look at the TRS-80 memory map, you will see that the beginning of the variables table is established immediately after the BASIC Program Text area in memory.

The BASIC interpreter keeps track of where the variables table begins by storing an address pointer in memory addresses 16633 and 16634. These two addresses contain the low- and high-order bytes of the address in memory at the beginning of the variables table.

To convert the two-byte number into the decimal memory location, do the following:

Multiply the contents of address 16634 by 256 and add to this product the value of the contents of address 16633. The resultant answer will be the address, in decimal, where the variables-table begins. After a BASIC program has been loaded, this address can be computed and displayed, using the following method:

In the command mode, from ready, type:

```
PRINT PEEK( 16634) * 256 + PEEK(16633) (Enter)
```

The number displayed is the address we are looking for. If you were to subtract 2 from this answer, you would have the address of the last byte of code for the resident BASIC program.

Whenever an @LOAD command is issued directly from the keyboard (command mode), this address pointer in 16633 and 16634 gets reset by the BASIC interpreter after the program has loaded. In consequence of this pointer being reset, all existing variables are lost.

If the @LOAD command is issued as a statement from within a program line, an important change in events occurs. In this case the ESF operating system intercepts the BASIC interpreter routine that resets the two-byte address pointer, after the new program has been loaded into memory.

Now you have a new BASIC program in memory, along with an already established variables table that can be accessed by the new program.

One other action occurs after the new program has loaded, the ESF operating system causes the new program to autoexecute.

This whole process is called program CHAINING. The net result of the chaining action is not only to call in a new program and still keep the previously established variables intact, but also to bring this new program in up-and-running.

If the new program is bigger than the original program then the chaining action will fail, as the old variables-table pointer will be overwritten by the new program. In this case you will get an Out of Memory error.

Thus, the first program loaded and run must be the longest program in the series of programs that are to be chained together. This ensures that the variables-table pointer always points to an address, high enough in memory, that it will not be overwritten by subsequentl programs.

If the first program to be run is not the longest program in the sequence to be called, then you must set the variables-table pointer yourself. This can be done with a couple of POKE statements, in the first line of the first program to be run. The method for doing this is as follows:

First, load the longest program in the sequence of programs that you must run. After it has loaded, PEEK at those two addresses, 16633 and 16634, and make a note of their contents. For example, let's say addresses 16633 and 16634 contain 240 and 82 respectively. Now insert the following statement as the first statement in the first line of the first program that must be run:

```
POKE 16633, 240: POKE 16634, 82: CLEAR n
```

Make this statement a permanent part of that first program and then @SAVE the subsequent files onto the wafer in the order in which you want them to be loaded. Now, every time that you run the first program, the variables-table pointer will be set to point just above the top of the longest program that you are going to use.

MAKING THE CHAIN ===

In summary, for proper program chaining action to take place, the following criteria must be met:

1. The program to be chained must be called by an @LOAD command from within the program. If the @LOAD command does not have a file number the next program on the wafer will be loaded.

2. The variables-table pointer must initially be set to point to an address high enough in memory, so that all programs to be chained have enough space in the BASIC program text area.

This pointer is automatically set to the correct value if the first program loaded and run is equal to, or is, the longest program to be chained. If the first program to be run cannot be the longest program, then the variables-table pointer must be altered by the first program. This is done by POKEing appropriate values into addresses 16633 and 16634. The appropriate values are determined by PEEKing these locations' when the longest program is in memory.

STRING VARIABLES & CHAINING ==

Passing numeric variables on to a chained program is of no concern, because the values for variables of this type always reside in the variables table, which is outside the program text area. String variables, however, need special consideration.

For a string variable to be passed on, its value must be residing in the string storage area of memory. If a string is defined as a constant within a BASIC program, then its value never gets put into the string storage area and it will be overwritten by the incoming BASIC program.

An example of a string being defined as a constant within a program would be:

```
10 A$ = "HELLO"
```

The value for A$ ("HELLO") would not be placed in the string storage area and therefore would be lost when the new program loaded.

Another way of defining strings that will prevent their being placed in the string storage area is to define them in DATA statements such as:

```
100 DATA "HELLO", "ESFOA"
```

The strings "HELLO" and "ESFOA", which would be assigned to some string variables by a read statement associated with that DATA line, would be lost as the new program loaded in and occupied the program text area.

The following are string-defining methods that will ensure that strings are placed in the string storage area:

1. Strings defined by INPUT and INXEY$ statements

```
10 PRINT "NAME PLEASE": INPUT A$
```

The value assigned to A$ in the above example would be placed in the string storage area.

2. Any time that any of the manipulative strings (CHR$, LEFT$, RIGHT$, MID$, STRING$) are used, their values are placed in the string storage area.

```
      10 A$ = CHR$(65)
      20 B$ = STRING$(10,57)
      30 C$ = "HELLO"
      40 0$ = LEFT$(C$,5)
      50 E$ = RIGHT$(D$,5)
      60 F$ = MID$(E$,1,5)
```

The values for A$, B$, D$, E$, and F$ in the above example would all be placed in the string storage area.

3.   If strings are concatenated (added together), their concatenated value will be placed in the string storage area.

```
10 A$ = CHR$(65) + CHR$(57) + STRING$(5, 46)
20 B$ = "HELLO " + "ESFOA"
30 C$ = "HAPPY "
40 0$ = "BIRTHDAY"
50 E$ = C$ + D$
60 F$ = "STRINGY" +
```

In the above example, A$, B$, E$, and F$ would all be placed in the string storage area. The F$ example in line 60 is an interesting and useful way to get a string that must be defined as a constant up into the string storage area. You simply concatenate a null string to the string constant. This will then move the value of the string constant up into the string storage area without altering its contents.

CHAIN YOURSELF UP ==

To get a little hands-on experience in computer bondage try keyboarding the following program. It will demonstrate the use of ESF commands in the execute mode as well as chaining.

Make sure that you have initialized the ESF, and then insert a blank certified wafer, then type the program in.

```
   10 CLS:
      PRINT "ADDRESS 16633 CONTAINS A";PEEK( 16633), "ADDRESS 16634
      CONTAINS A";PEEK( 16634):
      PRINT
   20 PRINT "THE STARTING ADDRESS OF THE VARIABLES TABLE IS";
      256 * PEEK( 16634) + PEEK( 16633):
      PRINT
   30 PRINT "X = " X: PRINT
   40 INPUT "Enter A NUMBER FROM 1 TO 3 "; X
   50 ON X GOTO 70, 80, 90
   60 IF X < 1 OR X > 3 THEN 10
   70 @LOAD 1
   80 @SAVE 1
   90 @NEW 2:
      GOTO 10
```

Let`s analyze the program first, before you run it.

Line 10 will clear the screen and display the contents of addresses 16633 and 16634. These will be the low- and high-order bytes that, when resolved, will be the decimal address of the point in memory where the start of the variables table will be.

Line 20 resolves this address from the above low- and high-order bytes and displays it to the screen.

Line 30 displays the value of variable X. It will, of course, be 0 when you first run the program.

Line 40 is an INPUT statement that asks you to Enter a number. When you Enter the number, its value will be assigned to the variable x.

Line 50 will jump program execution to line 70, 80, or 90, depending on which number you Entered. If you Entered 1, the program will jump to line 70, and so on, sequentially up through the number 3, where the program would have jumped to execute line 90.

Line 60 is a trap that will return you to the beginning of the program in case you get cute in answering line 40.

Lines 70, 80, and 90 contain ESF commands that will execute when those lines are accessed by the program.

Line 90 is a multiple-statement line in which program control will be returned to line 10 after the ESF command "@NEW 2" has completed its certify operation.

RUN THE DEMONSTRATION ==

Use the following steps as a guide in using the program. In this way, the purpose of this demonstration program will have been achieved.

1. Type RUN and press Enter.

Result ... The screen clears, and the contents of addresses 16633 and 16634 are displayed, along with the address to which these values resolve. Also on the screen will be displayed the value of variable X and a message instructing you to Enter a number from 1 to 3.

2. Type 2 and press Enter.

Result ... The number 2 is assigned to the variable X; and program execution jumps to line 80, whereupon this program is now SAVEd on wafer as file 1. After the verify cycle of the @SAVE operation is completed, program execution proceeds to the next line and the ESF executes a certify operation from the end of file 1 to the end of tape.

After the certify operation is completed by the @NEW 2 command, the program execution proceeds to the next statement, which, in this case, is in the same line. That statement directs program execution back to line 10, wherein the program will again display the information called for by lines 10, 20, 30, and 40.

Notice that this time the value of X has changed from 0 to 2. This is the result of your having Entered that number in answer to the displayed instruction "Enter A NUMBER FROM 1 TO 3".

3. You now have this program recorded on wafer as file 1. Before going any further, write down what the contents of addresses 16633 and 16634 are. Now answer the message instruction with the number 1.

Result ... The variable X is assigned the value 1. Program execution jumps to line 70 because of the action of line 50. At line 70, the program encounters an @LOAD command and executes it. The ESF operating system will immediately detect this as a chaining operation and will call in the designated @LOAD file number.

In this case, since the called program that is being chained is the same program as the one in memory, the program will not be any longer and proper chaining results should be observed. You should have noticed that the ESF immediately went into a read operation when line 70 executed and loaded with an autoexecute.

Upon executing, it again displayed the contents of 16633 and 16634, which, you will observe, remained unchanged. This proves that the chained program must have the same length as the calling program. Also, you will notice that the displayed value for the variable X is the number 1, which was the value assigned to that variable in the original calling program. This shows that the variable table was, indeed, unaffected by the @LOAD operation executed from a line statement.

BREAKING THE CHAIN ==

Now let `s do some experimenting with this program.

1. Change program lines 80 and 90 so that they LIST as follows:

```
80 @SAVE 2
90 @NEW 3:
   GOTO 10
```

2. Add a new line as follows:

```
100 REM THIS IS PROGRAM #2
```

This line was added simply to increase the length of the program.

3. Run the program, and answer the instruction message by Entering the number 2.

Result ... the same action as the original program run in steps 1 and 2, except that this time a slightly longer program was SAVED to wafer as a file 2.

4. Note that the contents of 16633 and 16634 have changed from the other program and that the resolved starting address of the beginning of the variables table is higher in memory. This shows that the ammended program now in memory is longer than the previous one. Write down the values displayed for the contents of 16633 and 16634.

5. Press the BREAK key so as to get out of the INPUT loop in line 40. Now, in the command mode, type "@LOAD 1" and press Enter.

Result ... File number 1 on the wafer will load. This is the unaltered original version of the demonstration program.

6. Change line 70 to read:

```
70 @LOAD 2
```

and run the program.

7. Answer the instruction message by Entering the number 1.

Result ... The program execution jumps to line 70, and the ESF trys to perform a chaining operation. The program crashes with an OUT OF MEMORY ERROR.

Reason ... File 2, which is the program being chained in, is longer than the program in residence and therefore cannot reside in the BASIC TEXT area whose upper boundary has been set by the values in 16633 and 16634 by the smaller original program.

8. Enter the edit mode for line 10, and insert at the beginning of that line two POKE statements for addresses 16633 and 16634 whose arguments are the values that you observed for the contents of those addresses when you ran program number 2. The beginning of line 10 should then read like this:

```
10 POKE 16633, nnn: POKE 16634, nnn:
cLS:
PRINT ... etc.
```

"nnn" in the above POKE statements represents where the values that you observed for those two addresses would be inserted.

9. After you are satisfied that line 10 is correct, run the program and observe that the displayed contents of 16633 and 16634 are what you just POKED in and not the lesser value that would have been there normally for this program.

10. Again answer the instruction with the number 1, and press Enter.

Result ... Because you have set the variables-table pointer to a value higher in memory by inserting those POKE statements, the action of chaining is realized. File 2 chains in and autoexecutes. The displayed value for the variable X is shown to be the number 1. This shows that the value for variable X has, indeed, been carried forward to the next program through the action of chaining.

CLEAR & CHAINING ===

A last thought to keep in mind while chaining is this: do not have any CLEAR statements in your chained programs. CLEAR, of course, will reset your variables table and you will not be able to access any of the variables that you had hoped to preserve through the action of chaining. Clearing of string space must only be done in the first executed program.

Sometimes, you may want to call in a program through a chaining operation only because you want to take advantage of the added feature of its giving you an automatic execute. In this case, a CLEAR statement might be appropriate, because you are not interested in preserving any variables. In fact, in a case such as this, it would be wise to include a CLEAR statement in your new program to prevent any initiated variables from being carried over and possibly causing erroneous results in the execution of the program.

This concludes the discussion on chaining. You should now be ready to use this powerful function in your own program development. One suggestion is to use the chaining feature when your complete program may be too long to reside in memory as one continuous program. You could then break up your program into modules and call in desired sections as they are needed, via chaining.

DATA FILES ===

All writing of data to tape is accomplished by grouping all related data together and saving that group of data to wafer as an entity. Identifying markers are recorded along with these groups of data so as to keep each group separate and distinct from any other group. These groups of related data are referred to as FILES.

Each file has recorded with it its own individual file identifier or number. In this way, any desired file may be called and read into the computer, or any file created within the computer may be saved with an identifying number to wafer.

There are two types of files that are of interest to us -- they are program files and data files.

Program files are the familiar sets of instructions that we load into the computer in order to cause the computer to create or produce a certain desired result. We normally refer to program files simply as "programs".

The desired result of running a program may range anywhere from having the computer solve a complex equation and print the answer to simply drawing a pretty picture on the screen. Another desired result may be the generation of certain data that may be used by that program itself, or possibly by another entirely different program. Any program whose primary purpose is to create and manipulate data is further classified as being a Data Base Management type of program.

Data that is created by this type of program can be preserved for future use, and saved to wafer as files, with identifier numbers. The action is similar to that of saving program files. These files of data are called, strangely enough, data files.

There is no data representing program instructions in a data file. A data file contains only code representing certain numerical and/or string data that will be used by a set of program instructions to create a desired result. In other words, the program instructions act on the data that is input from a data file.

Since the computer must know whether it is reading a data file or a program file, the format for writing and reading these files is different.

The ESF operating system and the Data I/O program take care of this special formatting for you automatically. As long as you use the ESF and Data I/O commands properly, all will be taken care of for you by the little black box.

For those who can’t rest until they know how the black box does its thing, there is an Advanced Programmers Guide, available through the Software Division of Exatron. In this guide, you will find source listings for the ESF operating system and the Data I/O program.

THE DATA I/O PROGRAM ==

Recorded as the first file on the programmed Starter Kit wafer is a 914-byte machine-language program titled "DATA I/O." The I/O stands for Input/Output, and the program allows you to read data in from a wafer and write data out to a wafer.

This program must always be resident in memory whenever you want to run an ESF version of a data-base program. With Data I/O in residence, you will have at your disposal five additional ESF commands, allowing you to program and manipulate data files.

These new commands are:

@CLEAR
@OPEN
@CLOSE
@PRINT
@INPUT

As mentioned previously, before you can run a database program, the Data I/O program must be in memory. In fact, it must be the very first program loaded. Only then can you load and run your data-base program. The reason for this is that the Data I/O program initially loads into the middle part of the computer memory area, and then automatically relocates to the top of 1024-bytes memory.

You can see that if a BASIC program were in residence before you loaded the Data I/O, it would be wiped out by the loading and relocating action of the Data I/O program. Most people make the Data I/O program the first file on a wafer and save their database program as file 2. In this way, the programs will be loaded in the proper order.

Several things happen when the Data I/O program relocates to the top of memory. Once it is in residence, it will cause the sign on message:

```
ESF DATA I/O VERSION 4.1A
```

to appear on the screen. This informs you that the five DATA I/O commands are now at your disposal.

The ESF Data I/O system is designed to write and read data in 256-byte blocks.

A 256-byte buffer area is used to store variables and data as they are written to, or read from, a wafer as data files. The Data I/O is now in protected memory, ready to accept data as generated and controlled by the Data I/O commands.

THE DATA I/O COMMANDS ==

Let’s start our discussion of the Data I/O commands with the command "@OPEN." We will first look at its use in setting up the system for writing data files and later its use in setting up the system for reading data files.

Assume for the purpose of the following discussion that the Data I/O has been loaded and that there is a blank, certified wafer inserted into the ESF. In this part, we will just define and discuss the action of the commands. Later, you can key in a demonstration database program and actually exercise these commands.

@OPEN ==

When the "@OPEN" command is used, this is referred to as "opening a data file"'.

The command "@OPEN" must always be affixed with a file identifier number. The reason and action is similar to why a file number must always be affixed to an @SAVE command when dealing with program files. If this is to be the first data file written to wafer, then the command would be written as "@OPEN 1."

The file identifier may be in the form of a decimal constant, a variable, or an expression. This will be the first Data I/O command that a data-base program will execute in any Data I/O operation. Later, when we discuss the @CLEAR command, we will find that there are occasions when that command might be the first to be executed.

If your computer system comprises only one ESF drive, then you can have only one file open at any one time. If you try to open more than one file on a single drive system, an FC ERROR will be returned.

When the program encounters the "@OPEN 1" command, two important actions occur. One action can be observed, but the other is transparent. What you will observe when this command is executed is that the ESF will turn on and drive the blank certified tape around until it finds file-mark 0, then it will turn off. At this time, control of the system is returned to the program. No program instructions will be executed during the file mark search.

The other action that occurred upon execution of this command was the completion of the link so that the Data I/O program can receive data into its buffer. The data that will be written into the buffer will come either from the database program itself or from a data file being read in from wafer; just which will be determined by the next Data I/O command that is executed.

@PRINT ==

When the "@PRINT" command is used, this is referred to as "writing a data file".

Assume that the next action that we want to occur is that of writing a data file to wafer. The next Data I/O command that should be executed would be @PRINT.

The @PRINT command will be followed by a list of variables and/or a list of expressions representing certain numeric and/or string data generated by the database program. Each variable and expression must be separated by a comma.

There is no set order required for @PRINTing this list. The list may contain a mixed bag of various precision numeric variables, string variables, and expressions. Following is an example of how the Data I/O portion of a program might list, using the commands discussed so far:

```
100 @OPEN 1
110 @PRINT A, J$(D), F% + GC%, L$, B#
```

Of course, the data, as represented by line 110, would have already been generated elsewhere in the program prior to execution of this Data I/O routine.

When line 110 is executed, the data represented by the contents of this line will be PRINTed (written) into the DATA I/O program buffer area. One of two things must occur before the Data I/O program will cause the contents of the buffer to be written to wafer: this will happen when the buffer becomes full (256 bytes of data), or when the program executes an @CLOSE command. When either of these two actions occurs, program control will shift to the Data I/O, the ESF will go into a write operation, and the buffer will write its contents to wafer.

@CLOSE ==

When the "@CLOSE" command is used, this is referred to as "closing a data file."

In the above example, one more line needs to be added to the program. Make it:

```
120 @CLOSE.
```

Now you have a workable Data I/O portion of a database program. Since you can have only one file open at any one time on any particular ESF drive, there will be only one @OPEN command associated with any @CLOSE command; therefore, a file number is never affixed to an @CLOSE command. A syntax error will result if you do that.

In the above example program, nothing would have been written to wafer until the @CLOSE command was executed, because there was not enough data in line 110 to fill the buffer. When the @CLOSE command is executed, it causes the contents of the buffer to be written to wafer; and as a last operation just before control is returned to the program, an "end of file 1'" marker is recorded. The Data I/O program knows which file identifier to record because of the number that was affixed to the @OPEN command associated with this particular I/O operation.

If there was more than enough data in line 110 to fill the buffer, then the writing operation to wafer would begin before the @CLOSE command was executed. For example:

Assume that there were 575 bytes of data represented by all the different types of variables in line 110. The following sequence of events would occur:

As soon as the @PRINT command was executed, the data, as represented by line 110, would begin filling the Data I/O buffer area. As soon as 256 bytes had been written into the buffer, the Data I/O program would begin writing this data out to wafer. When the buffer was empty, the ESF write operation would stop and another 256-byte block would be written to the buffer. Again, when the buffer was full, its contents would be written to wafer.

The last cycle of events to occur in this example would be for the remaining bytes of data to be written to the buffer and remain there until an @CLOSE command was executed. At this time, the last buffer dump to wafer would take place and an "end of file 1" marker would be recorded.

@INPUT ==

When the @INPUT command is used, this is referred to as "reading a data file."

When you are ready to access the data that you have stored on a data file, you do it through the action of this command. The Data I/O portion of your program that will access the stored data will look similar to the portion that wrote the data into a file. Taking our previous three line program as an example, the complete Data I/O portion of your data-base program might look something like the following:

```
100 @OPEN 1
110 @PRINT A, J$(D), F% + GG%, L$, B#
120 @CLOSE
200 @OPEN 1
210 @INPUT X, K$(R), H%, S$, C#
220 @CLOSE
```

Let’s start our discussion of the @INPUT command with line number 200.

The @OPEN 1 command in line 200 will accomplish the same results as the @OPEN 1 command in line 100: i.e., wind the data wafer tape around to the file marker 0 and set up the linkage to the buffer in the Data I/O program. The buffer is now ready to receive data from either the database program or the data wafer, depending on whether the next command it receives is an @PRINT or an @INPUT.

Since line 210 is the next program line to be executed, the buffer will receive data from the data tape because of the @INPUT command in that line.

The @INPUT command must be followed by a list of variables (no expressions allowed this time), of the exact same type and in the exact same order that they were written into the file. You may assign name designators different from the originals, but they must be assigned in the same order and type as the originals and be separated by commas.

If, for example, you were to exchange the positions of X and S$ in line 210, the program would crash with a TM Error; the reason being that those data items no longer match in type or sequence the sequential list of data items written to the data wafer by the first three lines of the example program. In the next section, you will be asked to key in this little program and experiment with it. At the conclusion of the next section, you will have a clear idea of what you can and cannot do in a database program.

When the @INPUT command, along with its variables list, is executed, the ESF will initiate a read operation and input the variables data into the Data I/O buffer. The action of the buffer on an @INPUT command is a little different from what it is when accepting data from an @PRINT command.

You will recall that in an @PRINT situation, the buffer does not cause any external action to occur until after it is full or until an @CLOSE command is executed. In an @INPUT command situation, the buffer will assign the variables to your program variables look up table when they are read in. It does not have to wait until it is full or have an @CLOSE command issued before the variables that are read in become part of your program. You will test this out in the next section.

As with every Data I/O section of a database program, the final Data I/O command should be an @CLOSE. Whereas in an @PRINT command situation you needed the @CLOSE command to dump the final buffer load to wafer and cause the "end of file marker" to be written, the use of @CLOSE in an @INPUT command situation is different. It is used primarily for setting up the program for another, future, Data I/O operation. The computer will return an FC Error if a file-opening operation is attempted while a previously used file remains open.

Remember, to initiate a Data I/O operation, you must first open a file with the command "@OPEN", followed by a file number. Since you can have only one file open at any one time in a single drive setup, an @CLOSE command should be issued after the completion of every @INPUT operation to ensure successful future Data I/O operations.

DATA FILE CAUTION ===

When the data file write operation has been completed, there is no VERIFY cycle of the ESF as in an @SAVE operation. Always make a second copy of your data files or include a little verify routine in your program that will read back the data file and verify it before you exit the program and lose your stored data.

The data contained in each 256-byte buffer dump are referred to as a RECORD. The complete amount of data that are written between the @OPEN command and the @CLOSE command is called a file. Therefore, a data file may contain from one to as many RECORDS as the wafer tape has room to hold.

Each record is separated by a gap of approximately 256 bytes. This is caused by a time delay in the Data I/O program that allows the ESF motor to come up to operating speed before buffer data is written to the tape. Also, there is data written on the tape during the data file write operation that is used as end-of file markers and other system housekeeping information.

This all adds up to much more tape being used for data-file storage than one would think from a simple observation of the data to be saved. One must always plan ahead when dealing with Data I/O. Make sure that the wafer tape is long enough to hold all the desired data, as well as the extra housekeeping data that will be written along with
it. One of the Data I/O demonstration programs that came on the programmed Starter Kit wafer goes into this problem quite thoroughly and gives some suggestions on how to prepare and keep track of data file storage.

@CLEAR

When the "@CLEAR" command is used, this is referred to as "clearing a data file."

The @CLEAR command is unique in that it is normally used external to the Data I/O access routine. When an @CLEAR command is issued, it will CLOSE any files that may be open. If there are no files open at the time that an @CLEAR command is executed, nothing happens other than that the program just continues on executing subsequent instructions.

You will normally find an @CLEAR command residing at the beginning of a program or as part of an error trapping routine, or both. There are two major differences between an @CIJEAR command and an @CLOSE command. Although both will close a file, the action in doing so is different.

An @CLOSE command will not only close the file that was OPEN but will also, if given in an @PRINT operation, cause the contents of the buffer to be written to wafer along with an "end of file" marker.

The @CLEAR command, however, closes the file but does not write the buffer to wafer. The @CLEAR command will also make the buffer area once again available for data storage, just as though it had been written out with an @CLOSE.

The @CLEAR command closes and clears ALL files and buffers - on ALL drives if you have a multi-drive system. Therefore, multiple files and buffers can be restored to a closed and cleared condition with just one command.

There are two reasons why you should include an @CLEAR command at the beginning of your program and/or in an error trapping routine. If the program crashes when files are open you certainly want all files to he closed, so that you may initiate your Data I/O routines again after correcting the error.

In addition - and most importantly - you don `t want any data and "end of file " markers being written on the tape after an error was detected. An @CLOSE command would of course do just that and could cause a situation where you could no longer access any file data.

Say you were writing data to a wafer and ran out of tape - a "TAPE TOO SHORT" error would be executed and the program would crash. If at this time an @CLOSE command were issued, the buffer would write its remaining contents to wafer along with its "end of file" marker. As you might surmise, since you were at the end of the tape when this error occurred, the data would simply be written right over the EOT/BOT splice and, in all probability, right on into the beginning of the recorded data of File 1.

The @CLEAR command can be used by itself, or it can be affixed with a number up to and including 8 (the number of ESF drives that can be connected at one time). The number may be in the form of a decimal constant, a variable, or an expression.

@CLEAR AFTER ERRORS! ==

NEVER close a file after an error with an @CLOSE command. Use the @CLEAR command instead. In writing database programs always include error trapping routines.

REMEMBER: Preventing loss of data is the number-one consideration. Use of @CLEAR in all error trapping routines and at the start of all programs is a golden rule.

@CLEAR N - THE SIDE EFFECTS

Normally, in a single drive system, the @CLEAR command will not have a number affixed to it. Affixing a number is normally reserved for use in the beginning of a program that will be operating with a multi-drive system. What it does, when executed, is set aside the indicated number of 256-byte buffers in memory for use by the same indicated number of ESF drives. For example, an @CLEAR 3 command would reserve three 256-byte buffers to be accessed by three different ESF drives.
Another effect of issuing an @CLEAR affixed with a number is that it will also reset the LEVEL II variables pointer, thereby causing all variables generated by the program to be lost. It goes without saying that you would not want to use an @CLEAR command with an affixed number in your error trapping routines.

If for some obscure reason you did want to use the @CLEAR with an affixed number in a single drive system, it would be written as @CLEAR 1. This would accomplish everything a plain @CLEAR command would do, along with the added neat feature of destroying access to any variables that you may have generated in the program. If this is what you want, go to it; otherwise, better just use @CLEAR by itself.

@CLEAR and @CLEAR 0 do exactly the same thing. Files are CLOSED with no loss of variables. It's when you use a number from 1 to 8 that you lose your variables. Another point to remember is that you will reduce your available memory by 256 bytes every time you increase that affixed number by one. This is because having that affixed number also commands the Data I/O program to reserve that number of 256-byte buffers.

The command "@CLEAR 1'" will cause no change in memory size, as the Data I/O program automatically reserves one 256-byte buffer upon initialization. Therefore, on a single drive system, never affix a number greater than 1 to an @CLEAR command. To do so is only to waste memory. Use an @CLEAR 1 only if you want to CLOSE a file and reset the variable table. A plain @CLEAR is the usual, normal way to use the command in a single drive system.

SUMMARY

1. The Data I/O program must be in memory to run any ESF data-base program, and it must be the first program loaded.

2. Every Data I/O program routine must start with an @OPEN command and terminate with an @CLOSE command. The @OPEN command must be affixed with a file number. The @CLOSE command must never have a file number.

3. The @PRINT command is used between an @OPEN and an @CLOSE command to write data to wafer. The data may be represented by variables and/or expressions and must be separated by commas.

4. The @INPUT command is used between an @OPEN and an @CLOSE command to read data from a wafer. The @INPUT command must be followed by a list of variables, separated by commas, in the exact order and type as were originally written into the file. Expressions are not allowed.

5. The @CLEAR command is used to CLOSE all files that may be OPEN. If an @CLEAR command is issued during an @PRINT operation, all files that are OPEN will CLOSE without writing the buffer contents to wafer. The @CLEAR command is used in error trapping routines and also may be found as a line statement command at the beginning of a program.

6. Affixing a number from 1 through 8 to an @CLEAR command will cause all program variables to be lost. @CLEAR numbers 2 through 8 are used in multi-drive systems. On a single drive system, never affix a number greater than 1. To do so is only to waste memory. Use an @CLEAR 1 only if you want to CLOSE a file and reset the variables table. A plain @CLEAR is the usual normal way to use that command in a single drive system.

When you have finished studying this section, we suggest that your next step be to run and analyze the Data I/O demonstration programs provided for you on the programmed Starter Kit wafer.

EXAMPLE DATA BASE PROGRAMS

In this section, you will get some hands-on experience with two simple data-base demonstration programs. You will be asked to key in these sample programs and perform some experiments with them.

First, make sure that the Data I/O program is loaded and that you have a blank, certified, 5-ft wafer in the ESF drive. If you have a multi-drive system, put the wafer in drive 0. At this time, these demonstration programs will be used only in a single drive configuration.

Please key in the following program:

```
10  CLS
20  A = 1:
    D = 5:
    J$(D) = "HELLO":
    F% = 3:
    GG% = 4:
    L$ ="ESFOA":
    B# = 159.87654
30  @CLEAR
100 @OPEN 1
110 @PRINT A, J$(D), F% + GG%, L$, B#
120 @CLOSE
130 PRINT "FILE WRITTEN":
    FOR T = 1 TO 1000:
    NEXT T
200 @OPEN 1
210 @INPTJT X, K$(R), H%, S$, C#
220 @CLOSE
230 PRINT "FILE READ":
    FOR T=1 TO 1000:
    NEXT T
300 PRINT:
    PRINTK$(R), S$, X, C#, H%
```

Let `s analyze the program first and then run it.

Line 10, of course, clears the screen; and line 20 establishes our variables.

Line 30 will close any files that may have been left open during a previous operation. The @CLEAR command placed at the beginning of a program is very useful. If during your program development you have a program crash with a file open condition, you don `t have to issue @CLOSE commands in order to run the program again. After you have debugged the problem, simply type RUN, and the @CLEAR command will close all files for you and allow the Data I/O routines of your program to execute properly.

Line 100 is the start of the Data I/O routine. The @OPEN 1 command alerts the Data I/O program that a Data I/O operation is about to be performed and that this will be the first data file because the @OPEN command has the number 1 affixed to it.

Line 110 commands the Data I/O program to accept into its buffer data as represented by the variables following the @PRINT command.

Note: The expression F% + GG% is calculated by the computer first and then written into the buffer as an integer variable.

Line 120 closes the file, causes the contents of the Data I/O buffer to be written to the wafer, and records an "end of file" marker. In this case, the file mark data will represent the number 1.

Line 130 prints a message on the screen and causes the computer to pause long enough for you to read it before continuing on with the next program instruction.

Line 200 causes the same result as line 100.

Line 210 commands the Data I/O program to begin a read operation and input data from file 1 of the data wafer. During this operation, the Data I/O program will continually check that the data being read matches the variable type that is defined by the list following the @INPUT command.

Line 220 closes the file and terminates the Data I/O operation.

Line 230 prints a message to the screen and allows enough time to read it before dropping through and executing the next line.

Line 300 is where the program acts on from the last Data I/O operation. In desired was simply to print the data to the
the data that was read in this case, the only action screen.

RUN IT & DEBUG IT

Now it `s time to have some fun. Run the program, and verify that it operates as described above. When you are sure that you have a bug free program, continue on with the following experiments.

1. Remove the File 1 designator from the @OPEN command in line

100 and run the program.

Result ... FC ERROR IN 100.

Reason ... An @OPEN command must have a file number.

2. Change line 100 to read "100 @OPEN 3". Run the program.

Result ... ESF runs continuously.

Reason... The ESF is searching for an "end of file 2"' marker to know when to start reading a file 3. Since there never was a data file 2 written, the ESF will search forever. Stop the fruitless search by pressing the BREAK key.

3. Change line 100 and line 200 to read

```
100 @OPEN 2

200 @OPEN 2
```

Run the program.

Result ... A data file number 2 was written and read.

Reason ... This step increased the @OPEN file number by one and resulted in a successful data-file write operation. This was because an "end of file 1" marker was on the tape from the initial run of this program just before you initiated step 1 of this series of experiments.

4. Type "@NEW 2", and press Enter.

Result ... The ESF goes into a tape-certify operation from the end of file 1 to the end of the tape. A byte-count message at the end of this @NEW 2 operation tells you how much room you have left on the tape from the end of file 1 for additional data storage.

Reason ... The file number 2 affixed to the @NEW command instructed the ESF, through its operating system, to begin the @NEW operation at the end of a file 1 marker.

5. Restore lines 100 and 200 to their original syntax; i.e., "@OPEN 1". Type RUN 200, and press Enter.

Result ... The program reads the data from data file 1 and prints it to the screen. This shows that the previous @NEW 2 operation did not affect data file 1.

Reason ... The @NEW write cycle terminates at the EOT/BOT splice and therefore does not overwrite any of the file 1 data.

6. Remove the first comma in line 110, and run the program.

Result ... TM ERROR in line 210.

Reason ... Removing the first comma in line 110 caused the first data item in that line to now become a string variable labeled AJ$(D). Line 210 is, of course, looking for the first data item from that data file to be a single-precision numeric variable; and it is ready to assign to it the label X. Instead, it reads a string variable at this point, and a Type Mismatch error occurs.

7. Replace that first comma, and remove the second comma in line 110. Run the program.

Result .. .SN ERROR in line 110.

Reason ... Removing the second comma in line 110 resulted in a syntax error, because you now have a variable labeled "J$(D) F%+GG%". This, of course, is not an allowed way of labeling a variable, and the Level II interpreter rejects it as improper syntax.

8. Reinsert the comma, and remove line 30. Run the program.

Result ... FC ERROR in line 100.

Reason ... The program crashed in step 7 while a file was open. Removing the @CLEAR command in line 30 prevented the file from being automatically closed when the program was run again by step 8. Attempting to open an already open file results in an Illegal Function Call error.

9. Replace line 30, and affix the number 1 to the @CLOSE command in line 120. Run the program.

Results ... SN ERROR in line 120.

Reason ... An @CLOSE command affixed with a number is not part of the command table of the Data I/O program. A syntax error is therefore returned.

10. Remove the number affixed to the @CLOSE command, and interchange the S$ and C# positions in line 210. Run the program.

Result ... TM ERROR in line 210.

Reason ... As in step 6, a mismatch in the type of variable expected to be read from the data file has occurred. The data read in from a data file must be assigned the correct type designator. The type designator assigned by the @INPUT command is determined by the sequence of variable types written to the data file by the @PRINT command.

11. Restore S$ and C# to their original positions in line 210. Remove line 200 and run the program.

Result ... FC ERROR in line 210.

Reason ... A file must be opened by an @OPEN command before an @PRINT or @INPUT command can be executed.

12. Restore line 200 and remove line 120. Run the program.

Result ... FC ERROR in line 200.

Reason ... Again, an attempt was made to open a file that was already open. When line 100 was executed, it opened file 1; but it was never closed before an attempt was made to execute line 200.

13. Replace line 120, and affix a number between 1 and 8 to the @CLEAR command in line 30. Run the program.

Result ... No variables data printed to the screen.

Reason ... Affixing a number from 1 through 8 to an @CLEAR command will reset the program variables table.

14. Change line 30 to read ... 30 @CLEAR 9
Run the program.

Result ... FC ERROR in line 30.

Reason ... The maximum allowable number that can be affixed to an @CLEAR command is 8. This corresponds to the maximum number of ESF drives that may be connected to your computer at any one time.

15. Restore the @CLEAR command in line 30 to its original syntax: i.e., @CLEAR Add a new line to the program as follows:

```
215 PRINT X, S$
```

Run the program.

Result ... The data represented by X and S$ were printed to the screen prior to the @CLOSE command being executed in line 220.

Reason ... On an @INPUT operation, the Data I/O program does not wait for the buffer to become full or an @CLOSE command to be executed before the variables read in are assigned to the program variables table. The Data I/O program will do this immediately upon the termination of a read cycle during an @INPUT operation. This cycle will terminate when either of the following two actions occur:

The read cycle will temporarily terminate when the buffer becomes full. This temporary termination will allow time for the full buffer to assign the data to the program variables table and reset itself for another read cycle. This action will repeat until all the data called for in the @INPUT command line has been read.

The second, and what would constitute a final, termination action is when the last data item has been read in by the @INPUT command. At this time, the buffer assigns the remaining variables to the program variables table and the program drops through to execute the next line.

16. Change lines 100 and 130 to read as follows:

```
100 @OPEN A

130 PRINT "FILE" A "written"
```

Add the following new line:

```
140 A = A + 1: GOTO 100
```

The program will now execute in the following manner:

When line 100 is executed, file A will be opened. Since the variable A was assigned the value of 1 in line 20, we have, in reality, opened file number 1. Lines 110 and 120 execute as before. Line 130 will print a message to the screen telling which data file number has just been written. Line 140 will increase the value of A by one and then go back to line 100 to repeat the data-file write operation again.

17. Run the program.

Result ... Each time program lines 100 through 140 execute, a successive data file is written to tape. This sequence of events will continue until there is not enough room left on the tape to write another complete file. At this time, the program will crash with a "TAPE TOO SHORT ERROR".

Reason ... Each time line 140 executes, it will increase the file number designator by one. Line 100 then opens this new, higher file number and writes the data to wafer as commanded by line 110. When the EOT/BOT reflective splice is detected by the ESF, its operating system will cause any write operation in progress to terminate and display the "TAPE TOO SHORT ERROR" message.

If this error occurs during a data-file write operation, as it did in this case, we are left with an open file to contend with. In an actual, full-fledged, data-base program, there would probably be an error-trapping routine that would close this file for us with an @CLEAR command.

Complete the following steps to close that last file left open by the program crash.

18. Type @CLEAR. Now, while watching the ESF, press Enter.

Result ... No action from the ESF.

Reason ... An @CLEAR command simply closes the file and resets the buffer to an effectively cleared condition without writing its contents to wafer.

19. Type RUN 200, and press Enter.

Result ... Data file 1 is opened by line 200 and read in by line 210. The contents of data file 1 were printed to the screen by line 300.

Reason ... Since issuing an @CLEAR command did not cause any buffer data to be written to wafer, file marker 0 and data file 1 remained intact.

20. Run the program again by typing RUN and pressing Enter.

The result and reason will be the same as for step 17.

21. This time, close the file, using an @CLOSE command. Type "@CLOSE", and - while watching the ESF - press Enter.

Result ... The ESF turns on with a write cycle, as evidenced by the write light turning on momentarily.

Reason ... There was still data in the buffer when the program crashed due to a TAPE TOO SHORT ERROR. When the @CLOSE command was issued, it caused the buffer to write its contents to wafer and then close the file.

NOTE ==

Sometimes the length of your data tape and the length of your data file can be such that a partial file will never get a chance to be written. The EOT/BOT splice can arrive at the exact moment that the last complete file was written. In this case, the tape will stop at the "TAPE TOO SHORT ERROR" message and the program would not have had enough time to execute another @OPEN command. The file would have already been closed, and typing "@CLOSE" would have not caused any further action.

If this happens on the wafer that you are using for these experiments, then you will not get to see the action of the @CLOSE command writing out the buffer. In order to observe this action, try adding a few more bytes to the data file by appending three or four more B* variables to the end of line 110. This should unbalance the situation enough for you to see an @CLOSE command dump the buffer. Don't forget to separate the added B# variables with commas.

22. Type RUN 200 and press Enter.

Result ... The ESF runs continuously, looking for the file 0 marker to know when to start the read operation for data file 1. Guess what! It's no longer there, because that last little bit of data that the buffer dumped to wafer overwrote that critical file-header information. The consequence is a lost file-i data file. Press the BREAK key to terminate the fruitless search.

23. Run the program again.

Result ... The ESF is in a continuous search mode.

Reason ... Because the file 0 marker was wiped out by the last-issued @CLOSE command, the ESF cannot find the correct position on the tape to begin writing data file 1. You will have to @NEW this tape once more before you can use it for writing data files or program files. That file 0 marker, recorded by an @NEW operation, must be on every tape before it can be used.

You might like at this time, before @NEWING the data tape, to replace the file number in line 200 with another, higher number and Enter a RUN 200. This will show that even though you lost file 1, the other files are still intact.

This completes our experiments on this little data-base program. As a final wrap up on understanding the basics of using Data I/O commands, take a look at the program listing on the last page of this section. It is very simple and is practically the same idea as the one that you have just been working with.

To test yourself, go through an analysis of it as we did for the one that you've been using. When you are sure that you know how it will run, go ahead and key it in and Run it. Compare what you see happen with what you thought would happen from your study of the listing.

If your analysis was correct, and I 'm sure it was, then you have successfully completed your first step towards learning ESF data-based programming. Admittedly this is a small step, but a very crucial and important one. From this point on, now that the basics are out of the way, learning to make use of the Data I/O commands in your own programming efforts will become much easier.

In the next part of this section, we will offer some additional suggestions on how you can increase your knowledge of ESF data-based programming.

LEARNING ABOUT DATA BASE PROGRAMMING =================================

The purpose of this short section is to point you in the right direction for learning more about data-base programming

Now, what `s next? If you`ve had no experience whatsoever in writing data-base programs, then the fact is you have many weeks and months ahead of you that will have to be devoted to study and experimentation. Those of you who are familiar with this type of programming should need only a little more time experimenting in the use of the ESF Data I/O commands, and you'll be on your way. The following comments and suggestions are directed towards those in the first category.

Data-base programming is certainly one of the more advanced fields in programming. College courses are offered in this subject. Taking some formal study at a local college is probably the best method for acquiring the knowledge necessary for becoming a good data-base programmer. If this can't be done, then you should seek out friends who are knowledgeable in the subject and ask for their assistance with the problems that you may be having.

One of the best places to meet and make friends with people of this type is at a local ESF Workshop. You will find a mixture of all computer talents attending these workshops on a regular basis, including knowledgeable people who are usually willing to help newcomers with an exchange of information and ideas. By doing your homework ahead of time, you'll make it easier for people to explain things and you won`t be wasting someone `s time with questions already covered in the manual. Still, when things aren't clear, don't hesitate to ask someone for help.

Visit your local library, book stores, and computer stores. There, you will find books covering almost any aspect of programming. Another excellent source of information is the many computing magazines. Subscribe to as many as you can afford. Sometimes you will find in an article just one little gem of information that is worth the price of the full subscription.

Another source of knowledge in various programming fields is analysis of programs written by others. Buy a data-base program and study it inside and out. Break it down and study it line by line. Try to understand just how the author accomplished certain results in the program. When it begins to become clear just how certain techniques were used, then start experimenting with the program by making changes in the code and observing the results. Pretty soon, you'll start seeing better ways to accomplish the same results and you `11 be on your way to originality.

You may try writing to the authors for help in understanding some difficult parts of their program flow. Most, I find, are willing to help if you include a self-addressed stamped envelope along with your question. Of course, there are those who will hide their code and will not share any of their knowledge with anyone. Nothing you can do here. They have their reasons for doing this, and you should simply seek help elsewhere.

So, for the novice who wants to get into data-base programming, it means a lot of study. Attend formal classes if you can, and by all means join a Workshop. Get books on the subject and read all pertinent magazine articles. Last--and very important--study the programming work of others.

Included on the Starter Kit program wafer are two Data I/O demonstration programs. They are slightly more advanced in coding techniques than the example programs given here. Expanded listings of the programs are included with the Starter Kit. We suggest that as a next step in gaining experience with the ESF Data I/O commands, you study these listings and RUN the programs. They were written to help you get acquainted with the way that ESF handles data. The comments in lines 0 through 3 of the expanded listing for the first Data I/O demonstration program will point this out.

When you fully understand just how the Data I/O commands are used in these demonstration programs, the next step should be for you to use and study the method of programming used in the "General-Purpose Data-Base Management Program." This program is also provided as part of the Starter Kit program wafer. This program differs from the demonstration programs in that this data-base program is the real thing; it `s not a demonstration program on the use of Data I/O commands. You can put this program to practical use, and yet it`s not too complex. By now you should have no trouble in analyzing the code and making modifications to suit your own needs. It's an excellent program to begin your adventure into the real-life world of data-base programming.

EXAMPLE DATA-BASE PROGRAM:

```
10 CLS
20 @CLEAR
30 A = 1234
40 A$ = STR$(A)
50 B = 5678
60 B$ = STR$(B)
70 F$ = A$ + 8$
80 PRINT "I WILL NOW OPEN A FILE BY ISSUING AN @OPEN 1 COMMAND."
90 FOR T = 1 TO 2000
100 NEXT T
110 PRINT
120 PRINT "SEARCHING FOR THE FILE 0 MARKER."
130 @OPEN 1
140 PRINT
150 PRINT "FOUND IT AND NOW I WILL @PRINT A, AS, B, B$, F$"
160 FOR T = 1 TO 2000
170 NEXT T
180 @PRINT A, A$, B, B$, F$
190 PRINT
200 PRINT "I JUST @PRINTED A, A$, B, B$, AND F$ AND WROTE THE EOF-1
     MARKER BY CLOSING THE FILE WITH AN @CLOSE COMMAND"
210 @CLOSE
220 FOR T = 1 TO 3000
230 NEXT T
240 PRINT
250 PRINT "I WILL NOW REOPEN THE FILE WITH AN @OPEN 1 COMMAND."
260 FOR T = 1 TO 1500
270 NEXT T
280 PRINT
290 PRINT "SEARCHING FOR THE FILE 0 MARKER."
300 @OPEN 1
310 PRINT
320 PRINT"FOUND IT AND NOW I WILL @INPUT THE VARIABLES DATA IN THE
    EXACT-SAME ORDER AND TYPE AS I @PRINTED THEM."
330 FOR T=1 TO 5000
340 NEXT T
350 @ INPUT C, C$, D, D$, G$
360 @CLOSE
370 PRINT
380 PRINT"I HAVE COMPLETED INPUTING ALL THE DATA TYPES AND HAVE CLOSED
    THE FILE. PRESS A KEY AND I WILL MANIPULATE THE DATA IN VARIOUS WAYS."
390 IF INKEY$ ="" THEN 390
400 CLS
410 E$=C$+D$
420 PRINT C
430 PRINT D
440 PRINT C+D
450 PRINT
460 PRINT VAL(C$), VAL(D$)
470 PRINT VAL(E$), VAL(G$)
480 PRINT VAL(G$)
490 PRINT
500 PRINT USING"####",VAL(C$), VAL(D$)
510 PRINT USING"########",VAL(E$), VAL(G$)
520 PRINT USING"########",VAL(G$)
```

MULTIPLE-DRIVE SYSTEMS ==

The ESF system is capable of addressing up to eight drives; therefore, you may have up to eight drives connected to your system at one time. Each drive is wired internally to represent a certain drive number, and no drive can be accessed unless it is addressed by its own particular number.

The ESF drive that comes to you in the Starter Kit is wired and designated as drive 0. The ESF operating-system ROM is installed in drive 0, and no other drive has this ROM; therefore, every ESF system must have a drive 0 connected in order for the system to operate.

Any additional drives that you connect should be sequentially wired and designated as drives 1 through 7. When ordering additional drives, always include the drive number with your order. There is no difference in appearance among different drives, therefore you should mark them in some way when you receive them.

Connecting additional drives to your system is accomplished in the same way that you connected your original drive 0, i.e., to a bus-extender cable. If there are no more connectors left on your existing bus extender, then you should order a new extender with the number of male connectors you`11 need to complete the installation.

Have all drives connected in sequential order, beginning with drive 0, outward from the first connector after the connection to the computer card edge.

TALKING TO MORE THAN ONE ESF ==

When you power on your system and initialize the ESF, all commands issued to the ESF will automatically address drive 0. In other words:
at ESF initialization time, the default drive is drive 0. It doesn't matter how many drives you have connected to the system, the default drive will always be drive 0; and every @COMMAND that you give will only affect this drive.

To address any drive other than the default drive, you simply insert a pound sign (#), and drive number after the "@" sign and before the command word.

The following examples will clarify this by showing the syntax required in an actual command situation. Assume for the purposes of the following discussion that you have a two-drive system and that it is connected and initialized, with a wafer in each drive.

If you were to issue a simple @NEW command at this time, then the wafer in drive 0 (default drive at power up) would be certified and no further action would take place until another command was issued. So far, the action is just as in a single-drive setup.

Now, if you wanted to certify the wafer in drive 1, you would alter the command slightly so as to address that drive while issuing the command. The proper syntax for this command would look as follows:

```
@ #1 NEW
```

Notice that the "#" sign and drive number are placed between the @ symbol and the command. If this command were now issued, the action would be as before with drive 0, except that now the wafer in drive 1 would be certified. spaces are not required between sections of the command. The command is shown this way only for clarity.

The ESF operating system will ignore any embeded spaces in the command.

That drive-number insertion into a command right after the @ symbol is the only difference between any command issued in a single-drive system and one issued in a mutiple-drive system. All else remains the same, and the commands may be issued in the command mode or the execute mode, just as before.

IMPORTANT - CHANGE OF DEFAULT DRIVE

Once a drive has been addressed correctly, it becomes the default drive - until another drive is addressed.

This means that once a drive has been specifically addressed to execute a certain command, you will not be required to address that drive again to have it execute further commands. A normal command without an inserted drive number is sufficient from that point on. Let's clarify that with an example.

Take the example command that we just issued:

```
@ #1 NEW
```

Upon completion of the certify operation, drive 1 will be the default drive and any command issued from that time on will be executed by that drive.

As an example, if we now wanted to SAVE a program to the wafer in drive 1, we would simply issue the normal @SAVE 1 command, and the program would be SAVED on that drive. If later you wanted to SAVE a program to drive 0, then you would have to address that drive with:

```
@ #0 SAVE 1
```

The default drive has now shifted back to drive 0, and all subsequent commands will be executed by that drive.

You can test a multiple-drive system in much the same way as a single-drive system, because of the default drive select feature.

DATA I/O AND MULTIPLE DRIVES ==

All of the information presented thus far applies equally to Data I/O commands. The only special requirement when dealing with data files is when you want to have multiple files open at the same time, on different drives.

In this case you must include an @CLEAR statement at the beginning of your program (before any variables get set), with an affixed number equal to the number of drives that will have files open at the same time. For instance, in our example case of a two-drive system, an "@CLEAR 2" command would be the correct syntax to use.

If you want to open a file and write or read data using a drive other than the default drive, you must specifically address that drive, just as in using the regular ESF commands. For example, to open a file on drive 1 and write data to it, the Data I/O conunand would be as in line 50 of the following example Data I/O routine.

```
50 @ #1 OPEN 1
60 @PRINT A$, X, Y
70 @CLOSE
```

Notice that the drive-number designator needs to be inserted only into the @OPEN command. All other Data I/O commands are issued in the ordinary way.

In the above routine, drive 1 has become the default drive, and any further Data I/O routines concerning that drive will not have to be addressed with an @ #1 OPEN 1 command; a simple @OPEN 1 command will now suffice. If, however, you want to open a file on any other drive, you must address that drive in the @OPEN command. For example, if you now wanted to open and read a file on drive 0, the proper syntax for this additional Data I/O routine in your program would be as follows:

```
80 @ #0 OPEN 1
90 @INPUT J$(D), F, R%
100 @CLOSE
```

Inserting a specific drive number into the @OPEN command and the use of a number affixed to the @CLEAR command are the only differences between Data I/O operations in a multiple-drive system and in a single-drive system.

MULTIPLE-DRIVE DEMONSTRATION PROGRAM =================================

At the end of this section is a small, simple program that when run and analyzed will summarize all the concepts learned about using the ESF. The program was written for a two-drive system, but the concepts are the same for any multiple-drive system.

Please key in the program; and when you are ready to run it, insert a certified 5-foot wafer in each drive. The following is a line-by-line analysis of the program.

Line 10 ... a conditional statement that will jump execution of the program to line 300 if M + X = 2. When a program is first RUN, all numeric variables are set to 0 by the computer; therefore this program execution will effectively start with line 20. We'll see later on that line 10 will come into play during a chaining operation.

Line 20 ... clears the screen and sets aside two 256-byte Data I/O buffers for use by a two-drive system in a data-base type of program to follow.

Line 30 ... establishes the variables that will be used by the program.

Line 40 ... prints a message to the screen.

Line 50 ... opens file 1 on the default drive, number 0. The @OPEN 1 command causes that drive to advance the wafer tape to the file-0 marker.

Line 60 ... prints a message to the screen and delays further program execution long enough for it to be read.

Line 70 ... writes a data file to the Data I/O buffer space.

Line 80 ... causes the Data I/O buffer to write its contents to the wafer in drive 0 and closes the file.

Line 90 ... prints a message to the screen with a delay.

Line 100 ... prints another message to the screen with a delay.

Line 110 ... opens file 1 on drive 1 and causes the drive to advance the tape to the file-0 marker.

Line 120 ... prints a message to the screen with a delay.

Line 130 ... writes a data file to the Data I/O buffer.

Line 140 ... causes the Data I/O buffer to write its contents to the wafer in drive 1 and closes that file.

Line 150 ... prints a message to the screen with a delay.

Line 160 ... prints another message to the screen with a delay.