1720A
Training Package

P/N 648287
January, 1982

Technical Bulletin

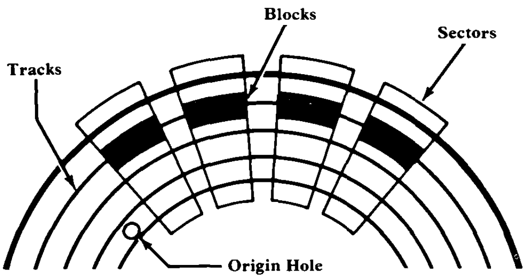
1720A Highlighted Learning Program

The Floppy Disk

The 1720A stores programs on a 5-1/4 inch floppy disk.

The disk itself is called the **MEDIA.**

The floppy disk has a smooth magnetic surface on which information is stored as sequences of magnetic pulses. The magnetic pulses are recorded along **TRACKS** on the surface of the disk.



The disk we use with the 1720A is **SINGLE SIDED** which means we can record data on only one side of the disk.

Our data is recorded using **DOUBLE DENSITY** recording techniques.

The disk is **SOFT SECTORED**, which means it has one origin hole.

It contains **35 TRACKS** and **350 BLOCKS.** Two **BLOCKS** are always used for the **DIRECTORY.** Each block stores 512 bytes (256 words) of data so the total storage capacity of the disk is approximately **175K BYTES (87K WORDS).**

Our files are stored on adjacent sectors so they are called **CONTIGUOUS FILES.**

A complete description of our program storage device is as follows:

The 1720A uses a 5-1/4 inch floppy disk. It is a **SINGLE-SIDED, DOUBLE DENSITY, SOFT-SECTORED 35-TRACK** disk. Its storage capacity is **175K BYTES (87K WORDS)**, and it has a **CONTIGIOUS FILE STRUCTURE.**

Because of the large amount of data stored on it, the floppy disk is also referred to as a **MASS STORAGE DEVICE.**

The Floppy Disk: Friend or Foe?

The floppy disk is **EASY TO USE** and allows **FAST ACCESS TO THE DATA** but if mistreated it can be your **WORST ENEMY.**

In other words, **DO NOT FOLD, SPINDLE OR MUTILATE!**

Care and Handling

- Always keep the disk in its protective cover when not in use.
- Careless handling of the disks may damage them. Avoid dropping, throwing or twisting the disks.
- Make sure to store disks vertically. Stacking may distort them and affect their contents by pressing the side covers into the disk.
- Direct sunlight may warp the disks. See that they are protected from the heat of the sun.
- Magnetic sources may distort data on the disks. Keep them away from electric motors, generators and transformers.
- Use a felt tip pen to mark disk labels. Pressure from a ball point pen or pencil may distort the data on the disk.

Loading the Disk

To load the disk, open the front disk entry latch and gently insert the disk label up, slot side first, into the disk entry latch using partial closures to seat the disk. Always grasp the disk by the cardboard cover and avoid touching the disk itself.

Note: If the door is closed when the disk is improperly seated, the disk center hole may be damaged. This can be prevented by partially lowering and raising the disk entry latch to seat the drive hub prior to total closure of the disk entry latch.

Formatting the Disk

- In order to write onto a blank disk it is necessary to go through a preliminary step called **FORMATTING.**

- During the formatting step, the floppy disk drive designates sectors and tracks using appropriate magnetic codes. This is an automatic process which the 1720A performs using the **FILE UTILITY PROGRAM (FUP).**
- The command for formatting is **/F.**
- Once the disk is formatted, it is ready to use. You can **WRITE** information onto it and **READ** it back.

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0076A-10U8003/SE EN

Technical Data

1720A Highlighted Learning Program B0077

Startup, Self Test, FDOS, Conmon

The 1720A controller goes through a **series** of **operations** at **start-up** to insure the **functional operation** of the **hardware** and to allow the **programmer/operator** to get into **BASIC** quickly and easily. Each operation is presented in the order in which it occurs.

Start-up follows a logical sequence which we will go through step-by-step.

The **SELF TEST** is loaded from the **ROM's (READ ONLY MEMORY)** located on the CPU board. We will follow the **START-UP** procedure step-by-step as detailed in the included flowchart.

START-UP
* Turn on power
* Insert disk

or press **RESTART** if disk is already mounted

The 1720A now goes through a **SELF TEST** routine.

SYSTEM SELF TEST — **FAILURE** — **ERROR MESSAGES**

The display on the 1720A will indicate;

Fluke 1720A Controller
HELLO
BOOT V x.y

The display now sequences to

Fluke 1720A Controller
SELF TEST
Error Message

SELF TEST performs a test on the following items:

***VIDEO BOARD**
***MEMORY BOARD**

***FLOPPY DISK INTERFACE**
***IEEE INTERFACE**

If any of the items do not test properly an **ERROR MESSAGE** will appear.

!VIDEO ERROR
!MEMORY ERROR
!FLOPPY ERROR
!IEEE MISSING OR FAULTY

If any **SELF TEST** errors are encountered, the display will also indicate

"Press any key to continue"

Pressing any key on the keyboard or Touch Sensitive Display will cause the start-up procedure to continue.

The display then sequences to

FLUKE 1720A CONTROLLER
Loading
ERROR MESSAGE

The 1720A now checks to see if a floppy disk is mounted. If so it checks to see if an operating system is present and if so it then loads **FDOS.**

If a floppy disk is not mounted or the operating system is not present on the floppy disk, the 1720A performs the same checks on the E-disk. If the operating system is found first on the E-disk, then the E-disk becomes the system device. If the operating system is found first on the floppy disk, then the floppy disk becomes the system device. If any errors are detected, one of the following error messages will appear:

?FLOPPY ERROR
?DISK NOT MOUNTED

?NO SYSTEM ON DEVICE
?ILLEGAL DIRECTORY

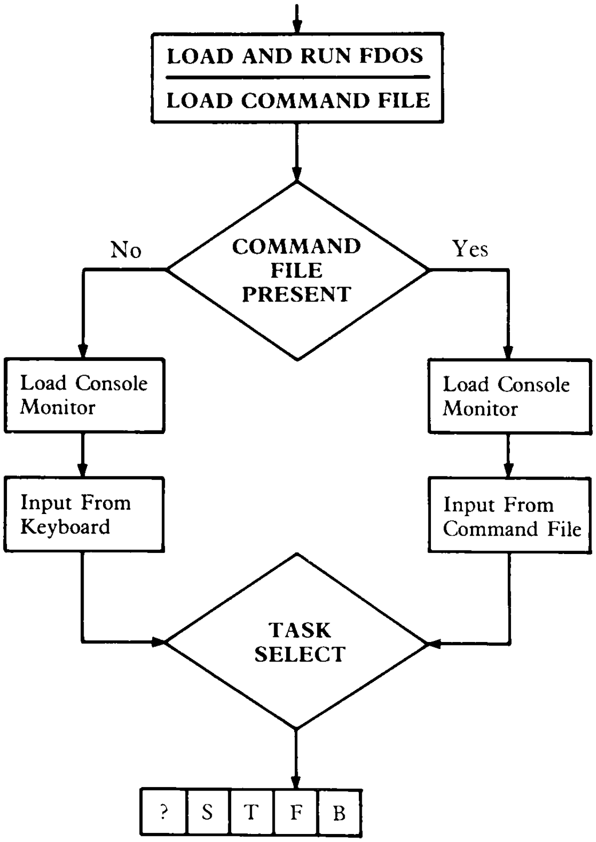
along with the message **"Press any key to continue."**

To continue after an error is encountered (such as **"?DISK NOT MOUNTED"**) and the error is corrected (mounting the disk), press any key on the keyboard or Touch Sensitive Display to continue the procedure.

FDOS IS NOW RUN (FLUKE DISK OPERATING SYSTEM)

FDOS manages disk operations and oversees communication between the disk and the Controller.

FDOS also checks for the presence of a **COMMAND FILE.**



The **COMMAND FILE** stores a set of instructions that normally would be input by the operator from the keyboard before **BASIC** is called and the **MAIN PROGRAM** is executed. It executes the preprogrammed instructions and can call up **BASIC** and run the main program. It allows, therefore, a set of start-up instructions to be executed automatically from the disk without the use of a keyboard. The **COMMAND FILE** will be covered in detail in a later lesson after keyboard execution of these instructions is learned. If no **COMMAND FILE** is present the 1720A now loads and runs **CONMON.**

CONMON stands for **CONSOLE MONITOR.** This is the program that oversees communication between the user and the 1720A. It allows selection of various utilities and programs. It asks the user to respond to the prompt character #.

The selections Are:

?displays a menu of the selections.
S SET BAUD RATE UTILITY
T SET TIME/DATE UTILITY
F FUP (FILE UTILITY PROGRAM)
B BASIC INTERPRETER

At start-up, the baud rate defaults to 4800 baud. (Internally selectable) The **SET BAUD RATE UTILITY** allows the baud rate on the two RS-232 ports (KB1, KB2) to be changed to the desired baud rate. An example of how to set the baud rate is shown below:

*The underlined portions indicate the user responses.
*(CR) stands for carriage return.

Example:

```
Console Monitor Versions x.y  11:08  26-SEP-79
# S (CR)
SET 11:08
SET Version x.y
Enter baud rate:
KB1 (J23):  2400 (CR)
KB2 (J22):  110 (CR)
```

After the KB2 baud rate is entered, the 1720A returns to **CONMON** and displays:

```
Console Monitor Version x.y  11:00  26-SEP-79
#
```

The **SET TIME/DATE UTILITY** sets the internal clock of the 1720A. Once it is set, it is supported by battery back-up even though power is removed from the 1720A. The procedure is shown below:

```
Console Monitor Version x.y  11:08  26-SEP-79
# T (CR)
TIME  11:08  26-SEP-79
Enter date:  DD-MM-YY  27-9-79 (CR)
Enter time:  HH-MM  11-14 (CR)
```

After the time is entered the 1720A returns to **CONMON** and displays the prompt #.

The **FILE UTILITY PROGRAM (FUP)** is a file transfer and management program. It is a versitile and often used program and will be covered separately in a later lesson. **FUP** allows operations such as listing the file directory, saving, merging, deleting, copying, and renaming files. It is also used to format and pack a disk. The procedure for calling **FUP** is shown below:

```
Console Monitor Version x.y  11:08  26-SEP-79
# F (CR)
FUP  11:08  26-SEP-79
File Utility Program Version x.y
*
```

To exit from **FUP** back into **CONMON**, use the procedure shown below.

Example:

`* /X (CR)` or `* CTRL P`

CTRL P means holding down the CTRL button and pressing the **P** button.

In order to **RUN** a **BASIC PROGRAM**, it is necessary to call up the **BASIC INTERPRETER**. It is called up from **CONMON** and is loaded into **MAIN MEMORY**. We can then **LOAD** the **USER'S BASIC PROGRAM** into **MAIN MEMORY** and **RUN** it. This will be covered in detail in a later lesson. The procedure for calling **BASIC** is shown below:

Example:

```
Console Monitor Version x.y  11:08  26-SEP-79
# B (CR)
BASIC  11:08
BASIC Version x.y
Ready
```

There are two ways to run a **BASIC** program. One way is to **LOAD** the program and then **RUN** it.

Example:

```
Ready
OLD "METER" (CR)
Ready
RUN (CR)
```

The other way is to **LOAD** and **RUN** the program with one command.

Example:

```
Ready
Run "METER" (CR)
```

In order to **HALT** a program that is running, the operator can enter the following:

```
CTRL C
Ready
```

A program can also be halted using the **ABORT** button on the front panel. This button **HALTS** the program and issues a **DEVICE CLEAR** over the **IEEE-488 BUS.**

Another way to **HALT** a program is to use the **CONTROL P** function. This function **EXITS** from **BASIC** and returns to **CONMON** (unless the program had been **SAVED** on the **DISK** or **E-DISK**, it would be lost).

Example:

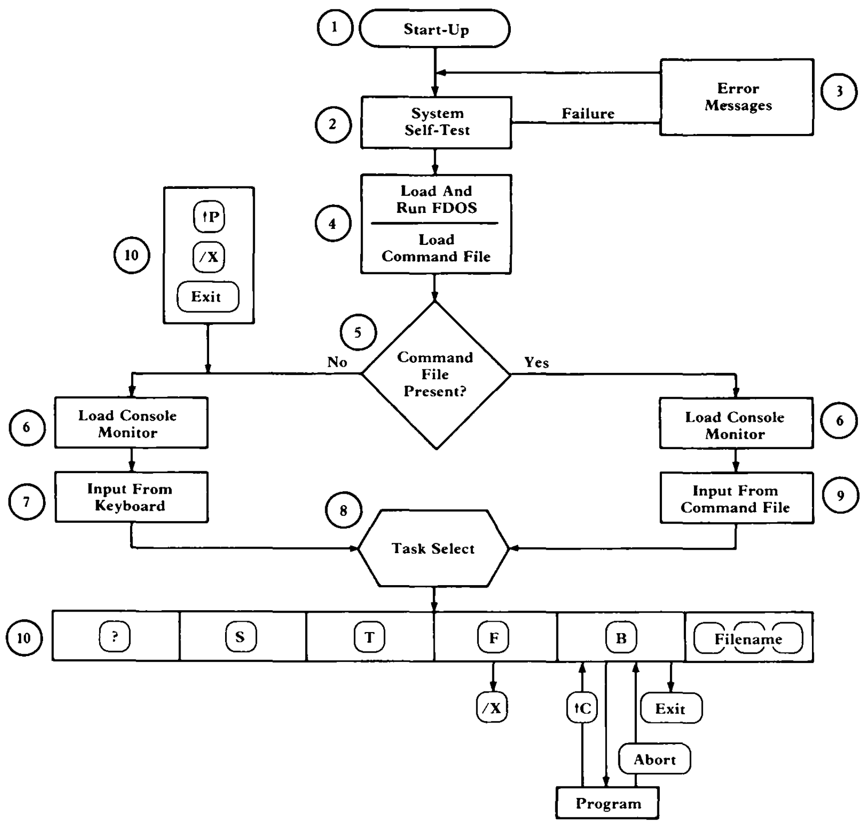
```
CTRL P
Console Monitor Version x.y  11:08  26-SEP-79
#
```

Another way to **HALT** the program and **EXIT** from **BASIC** is shown below.

```
CRTL C
Ready
EXIT (CR) or CTRL T
Console Monitor Version x.y  11:08  26-SEP-79
#
```

Using **CTRL T** instead of a **(CR)** will also cause a **(CR)** to be returned but in addition will **CLEAR** the **SCREEN** and **HOME** the **CURSOR.**

START-UP, SELF TEST, FDOS, CONMON, FLOW CHART



TASK	DESCRIPTION	EXITS
?	Displays a 'menu' of the various tasks performable.	None Required
S	Set Baudrate Utility	None Required
T	Set Time/Date Utility	None Required
F	File Utility Program	CTRL P. /X
B	BASIC Interpreter	CTRL P. EXIT
Filename	Uses any of the Assembly Language Programs	As Programmed

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box 43210, Mountlake Terrace, WA 98043
800-426-0361 (toll free) in most of U.S.A.
206-774-2481 from AK, HI, WA and Canada
206-774-2398 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0077B-10U8203/SE EN

Technical Data

17XXA Highlighted Learning Program B0078

FUP Fundamentals

Overview

FUP.CIL is a **file transfer** and **file management** program. This program is stored on the disk in **binary** and is called up from **CONMON.** The "**.CIL**" extension stands for **CORE IMAGE LOAD** and indicates that **FUP** is not a **BASIC** program. **FUP** is a versatile and often used program which provides the capability of performing **utility operations,** on **file structured devices,** which are not possible in the **BASIC** environment. The full list of options which can be performed by **FUP** are listed below:

COMMAND OPTION

1.	**?**	Displays a list of the FUP options
2.	**/A**	Assign the default system device (SYO:)
3.	**/B**	Binary file transfer
4.	**/D**	Delete a file
5.	**/E**	Extended directory listing
6.	**/F**	Format a file-structured device (floppy disk or E-Disk)
7.	**/L**	List the directory
8.	**/M**	Merge ASCII files
9.	**/R**	Rename a file
10.	**/S**	Search for bad blocks
11.	**/W**	Whole or partial copy
12.	**/X**	Exit from FUP
13.	**/Z**	Zero a file directory
14.	**/P**	Pack a file-structured device
15.	**/T**	Turn off error checking

In this lesson we will cover some of the **syntax** and **fundamental considerations** which are necessary in order to implement the **FUP COMMAND OPTIONS.**

FILENAMES and **EXTENSIONS** - Information such as **programs, data** or **text** which is stored on the disk (or E-Disk) is **accessed** via a **file name** with its associated extension.

NAME.EXTENSION

The **FILENAME** consists of from 1 to 6 letters, numbers, spaces or $ signs. The **extension** consists of from 1 to 3 letters, numbers, spaces, or $ signs.

Examples: "b" is used to show the space character

TEMP.BAS	FDOS.SYS	8502A.BAS
RESULT.DAT	FUP.CIL	8502A.BAL
RESULT.BIN	8502A.488	RESULT.b

There are six programs in the current operating system and an optional **COMMAND FILE.** These programs are:

FDOS.SYS	FLUKE DISK OPERATING SYSTEM
BASIC.CIL	BASIC IMMEDIATE & EDITING MODES
FUP.CIL	FILE UTILITY PROGRAM
CONMON.SYS	CONSOLE MONITOR
TIME.CIL	CLOCK & CALENDAR SETTING ROUTINE
SET.CIL	SET RS-232 BAUD RATE
COMMND.SYS	COMMAND FILE

The above **FILENAMES** and **EXTENSIONS** relate to the program stored under that **FILENAME.** The extension "**SYS**" is an abbreviation for **SYSTEM**; "CIL" is an abbreviation for **CORE IMAGE LOAD.** The system has been programmed to look for and recognize these specific **FILE NAMES** with the specific extension.

Note: *The system has not been programmed to prevent the programmer from using any of the above names with its extension to store other programs or data, and in so doing, delete the original system program!*

Every **FILENAME** gets an **extension** when it is used to **store** a program or data. The sytem assigns a **default extension** if the programmer does not use an extension.

Examples of **extensions and their origin:**

.BAS	this extension is assigned by default to any **FILENAME** which is used with a **SAVE** command in the **BASIC MODE** to store a **BASIC** program as **ASCII data. FUP** will assume a **.BAS** extension for any **FILENAME** entered without an extension.
.b	a ".space" extension is used as a default extension when a **FILENAME** is used with an **OPEN statement to store data, when the** program omits the extension.

.BIN	used by some programmers to indicate **binary data.** This is not a default extension. It has no special meaning for the system. Assigning an extension to a **BINARY** file is a good way to indicate it is not an **ASCII** file. Manipulation of **BINARY** files in **FUP** requires the **/B COMMAND**; therefore, the programmer needs to be able to recognize **BINARY** files.
.488	can be assigned to either a program or data file to indicate to the user its use with the IEEE-488 BUS. This is not a default extension; it has no special meaning to the system. "**.488**" may be used in place of "**.BAS**" and the system will still recognize the program as a **BASIC** program stored as **ASCII data**. "**.488**" may not be used in place of "**.BAL**". "**.DMO**" for demo, "**.CAL**" for calibration, "**.TMP**" for temporary, and "**.TST**" for TEST are all examples of how a group of programs could be associated by a common extension. These extensions are treated by the system the same as "**.488**".

Although **FILENAME** and **EXTENSION** are treated separately in the above discussion, the full file name of a program or data is really the **FILENAME** and the **.EXTENSION.**

*Note: In **FUP** only **ASCII** files may be listed out to a printer. If a file is stored on the disk in both the **.BAS** and **.BAL** versions (for example: **METER.BAS** and **METER.BAL**), and no extension is specified when using BASIC as in **RUN "METER"**, the **.BAL** file is used. If no extension is specified when using FUP, however, the **.BAS** file is used.*

Devices the 1720A can transfer data to or from any of **six I/O DEVICES** which are designated by a **two-letter** specification plus a **single number** indicating the unit number. The designations are listed below:

KB0:	Console device (keyboard/display)
KB1:	(J23) RS-232 port
KB2:	(J22) RS-232 port
MF0:	Floppy Drive
ED0:	Electronic Disk
MM0:	Main Memory (used for copying files)

To make operation easier, there is one other device designation **SY0:**, the **SYSTEM DEVICE**. Either the floppy drive (**MF0:**) or the Electronic Disk (**ED0:**) can be assigned as **SY0:**. The 1720A loads and runs the operating system from it.

The **SYSTEM DEVICE** can be assigned from **FUP** using the /A command as shown below. The **FUP** prompt character is an *.

MF0:/A	assigns MF0: as SY0:
ED0:/A	assigns ED0: as SY0:

SYNTAX of FUP COMMANDS

Now that we know what **FILENAMES, EXTENSIONS** and **DEVICES** are, we will show how these are used to perform operations in FUP.

The general FUP command is as follows:

```
The format for either the output or input file is:
[<DEVICE> ] : <FILENAME> [<EXTENSION>]
<OUTPUT FILE> [, <output file>] =
<INPUT FILE> [, <input file>] . <OPTION>
```

While multiple output files and input files may be specified (the designations in brackets [ ] are optional), we will work with the terms not enclosed in brackets as these are the minimum necessary components of a command line. This reduces the command line to:

```
<OUTPUT FILE>=<INPUT FILE>/<OPTION>
```

The **REFERENCE POINT** for all **FUP** operations is the = sign

OUTPUT FILE = INPUT FILE

Direction of Data Transfer

DATA is always transferred from the **RIGHT (INPUT** or **source FILE)** side of the = **sign** to the **LEFT (OUTPUT** or **destination FILE)** side

The default states are:

```
<DEVICE> = SYO
<EXTENSION> = .BAS
```

When **NON-FILE STRUCTURED DEVICES** like **RS-232 PORTS** and the **CONSOLE DEVICE (KBO:)** are used, a **FILENAME** and **EXTENSION** are not required. For example, **KB1:** is a valid description of an **OUTPUT FILE.**

We will now go through the use of the **FUP** commands. This command (?) displays the contents of the file **FUP.HLP** which summarizes the proper use and options of **FUP.** The command sequence is shown below:

/A	Assigns System Device	ED0:/A or MF0:/A
/B	Binary File Transfer	MF0:FUP.CIL=MM0:FUP.CIL/B
/D	Delete a File	TEST.TMP/D or ED0:TEST.1, TEST.2,TEST.3/D
/E	Extended Directory Listing	MF0:/E or ED0:/E or /E
/F	Format a Disk or the E-Disk	MF0:/F or ED0:/F
/L	List a Directory	MF0:/L or KB1: ED0:/L
/M	Merge ASCII Files	TEST.NEW=TEST.1, TEST.2, TEST.3/M
/P	Pack a Device	MF0:/P or ED0:/P or /P
/R	Rename a file	TEST.1=TEST.OLD/R
/S	Search for Bad Blocks	MF0:/S or /S
/T	Turn off error checking	EDO:TEST.BAD/T MF0:TEST.BAS/T
/W	Whole Copy or Partial Copy	ED0:=MF0:/W or MF0:=MM0:BASIC.CIL/W
/X	Exit from FUP	/X
/Z	Zero a Directory	ED0:/Z or MF0:/Z or /Z

The FUP program is accessed from the **CONSOLE MONITOR** (CONMON) program, using the following instructions:

When "#" is displayed
1. Type F
2. Press RETURN

In order to leave **FUP** and return to the **CONSOLE MONITOR:**

When * is displayed -
1. Type /X
2. Press RETURN

It is good programming practice to have a second disk and even a third disk with back up copies of your programs. FUP is a powerful tool and its operations are not reversible. If you inadvertently exchange the positions of the source and distination files in a FUP command, you could lose the source file, and in this case, a back up copy prevents a total loss.

John Fluke Mfg. Co., Inc.
P.O. Box 43210, Mountlake Terrace, WA 98043
800-426-0361 (toll free) in most of U.S.A.
206-774-2481 from AK, HI, WA and Canada
206-774-2398 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.
B0078B-10U8102/SE EN

FLUKE®

Technical Data

17XXA Highlighted Learning Program B0079

Communication Over The IEEE-488 Bus

IEEE-488

The concept of the **IEEE-488 Bus** allows **MULTIPLE INSTRUMENTS** to be connected to a **COMMON Bus.** Each instrument is assigned a **DEVICE ADDRESS** to allow **SELECTIVE COMMUNICATION** over the bus.

The **PRINT** and **INPUT** statements are the heart of bus communication. They allow **2-WAY COMMUNICATION** to take place between a controller and the various instruments connected to the bus.

A device which is **SENDING DATA** over the bus is called a **TALKER.**

A device which is **RECEIVING DATA** over the bus is called a **LISTENER.**

The general form of the **PRINT** and **INPUT** statements is shown below:

PRINT @	**2,**	**"COMMAND STRING"**
Makes addressed instrument a listener	Device Address	User assigned Command String enclosed in quotes
INPUT @	**2,**	**VARIABLE**
Makes addressed instrument a talker	Device Address	User assigned variable for storage of data Example R, A$, T%

An important rule of bus communication is that while there may be **MULTIPLE LISTENERS** on the bus, only **ONE TALKER** may be enabled at any given time.

We will now go through several examples to illustrate the fundamentals of communication over the bus.

In order to do this it will be necessary to use several additional bus commands.

INIT this command initializes the bus to a known state

CLEAR@ the selected device is addressed as a listener and a selective device clear is issued

WAIT Program execution is suspended for the time interval (in milliseconds) specified

Example 1: PROGRAM an 8502A to take a **SINGLE READING** and **DISPLAY** it on the controller.

```
10 INIT PORT 0          initializes bus
20 CLEAR@2              resets DVM
30 WAIT 5000            waiting to reset
40 PRINT@2, "VR2T0,"    programs DVM for 10V DC range
50 PRINT@2, "?"         triggers reading
60 INPUT@2, R           puts reading from DVM into variable R
70 PRINT R              displays reading on CRT
```

Example 2: MODIFY the above program to take **10 READINGS** and **DISPLAY** them in **REAL TIME**

add the following program lines to the Example 1 program

45 FOR I% = 1% TO 10%
75 NEXT I%

the complete program is shown below:

```
10 INIT PORT 0
20 CLEAR @ 2
30 WAIT 5000
40 PRINT @ 2, "VR2T0,"
45 FOR I%=1% TO 10%
50 PRINT @2, "?"
60 INPUT @ 2, R
70 PRINT R
75 NEXT I%
```

FOR NEXT LOOP REPEATS LINES 50, 60, 70 10 TIMES

The next example illustrates the important distinction between displaying the readings in **REAL TIME** (Example 2) and displaying them **AFTER** the measurement cycle is complete (Example 3).

EXAMPLE 3: MODIFY the program in Example 2 to **TAKE 10 READINGS** and **STORE** them in **VARIABLE R.** After the **LAST READING** is completed, the program **WAITS 3 SECONDS** and then **DISPLAYS** the **READINGS.**

Add the following program lines to the Example 2 program:

```
5 DIM R (10)
60 INPUT @ 2, R (I%)
70 ! DELETE THIS LINE FROM EXAMPLE 2
80 WAIT 3000
90 FOR I%=1% to 10%
100 PRINT I%; R(I%)
110 NEXT I%
```

The complete program is shown below:

```
5 DIM R(10)                          Dimensions R to hold up to 11 readings
10 INIT PORT 0
20 CLEAR @ 2
30 WAIT 5000
40 PRINT @ 2, "VR2T0"
45 FOR I% = 1% TO 10%                for next loop to take 10 readings
50 PRINT @ 2, "?"
60 INPUT @ 2, R (I%)
75 NEXT I%
80 WAIT 3000                         inserts delay before displaying readings
90 FOR I% = 1% TO 10%
100 PRINT I%; R (I%)                 for next loop to display 10 readings
110 NEXT I%
```

Simple as these programs are, the following items were accomplished:

1. Initialize the Bus
2. Clear an instrument
3. Send a command string to an instrument
4. Trigger a reading
5. Store readings in a variable
6. Display readings on the CRT
7. Use a wait statement
8. Dimension a variable
9. Use a for-next loop with an integer variable
10. Use a semicolon to separate printed variables

Note: Additional statements such as RBIN, RBYTE, WBIN, and WBYTE are available to transfer binary data, to speed up data transfer, to overcome data format problems and to control the sequence of data transfers. The software bulletin on the 8500 series digital voltmeter has examples of RBYTE and WBYTE for binary readings and their conversion to floating point format.

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A

B0079B-10U8207/SE EN

FLUKE ® **Technical Data**

1720A Highlighted Learning Program B0097

The System Command File

The **System Command File** stores a set of **initialization** instructions that normally would be input from the keyboard after applying power to the controller. It executes the preprogrammed instructions and can call up **BASIC** and run the **main program.** It allows therefore, a set of **start-up** instructions to be executed automatically from the disk.

Use of the **Command File** then, allows the 1720A to be operated completely **independent** of the keyboard.

Some typical **tasks** that would be performed by a **Command File** are shown below.

- CHECK TO SEE WHETHER THE CLOCK HAS BEEN DISTURBED
- SET THE BAUD RATES ON THE RS-232 PORTS
- FORMAT THE ELECTRONIC DISK
- ASSIGN THE SYSTEM DEVICE
- DOWNLOAD FILES FROM THE FLOPPY DISK TO THE ELECTRONIC DISK
- CALL UP BASIC
- EXECUTE IMMEDIATE MODE BASIC COMMANDS
- RUN A BASIC PROGRAM

All modes of the controller are accessible to a **Command File** just as they are from the keyboard.

There are **two ways** to generate a **System Command File:** It can be entered directly from the **keyboard** while in **FUP**, or a **BASIC program** can be used to read **data statements** into the file. Using the **BASIC program** allows the command file to be easily modified using the **BASIC editor.** For simple command files the **FUP** method is quicker. However once a command line is entered it cannot be corrected unless the whole command file is rewritten.

Generating A System Command File Using FUP:

1. From **CONMON** enter **"F"** to select the **File Utility Program.**
2. Enter **MFO:COMND.SYS=KBO:** This will assign the file **"COMMND.SYS"** on the floppy disk **(MFO:)** as the destination for source information to come from the keyboard.
3. Enter each **command** to be **stored** on a **separate line.** Use the **delete** key to correct for errors before terminating the line with a (CR).
4. Terminate the file with a **CTRL/2** (end of file).

AN EXAMPLE IS SHOWN BELOW:

Console Monitor Version x.y 8:52 19-JUN-80

```
* F (CR)
```

FUP 8:52 19-JUN-80
File Utility Program Version x.y

```
* MFO:COMMND.SYS=KBO:
T        (Calls TIME)
F        (Calls FUP)
EDO:/F   (Formats E-Disk)
Y        (YES)
/X       (Exit from FUP)
B        (Calls BASIC)
RUN"METER.BAS"
  Z      (End of File)
```

Generating a System Command File Using BASIC

This program creates two files on the floppy disk: a Command File under the name **COMMD. SYS,** and a copy of this program under the name **COMMD.BAS.** The DATA statements (except line 10) contain the actual command file. Use as many DATA statements as needed. Each **data** statement must contain **one** legitimate keyboard command. All modes of the 1720A are accessible to a Command File just as they are from the keyboard.

1. From **CONMON**, type **B** to select the BASIC Interpreter.
2. Type **EDIT** to select the **edit mode** of BASIC.
3. Now enter the following program using the **editing capabilities** of BASIC. Use as many **data statements** as needed, with one quoted command line in each statement.

```
1000    ON ERROR GOTO 10060
1010    DATA "    (COMAND       "
1020    DATA "    Inputs        "
1030    DATA "    For           "
1040    DATA "    CONMON        "
1050    DATA "    SET           "
1060    DATA "    FUP,          "
1070    DATA "    BASIC         "
1080    DATA "    ETC.)         "
.
.
(More lines of DATA, as needed)
.
.
10000   DATA "END"
10010   OPEN "MFO:COMMND.SYS" AS NEW FILE 1
10020   READ A$\IF A$="END" GOTO 10040
10030   PRINT #1,A$\GOTO 10020
10040   CLOSE 1
10050   KILL "MFO:COMMND.BAS"\SAVE "MFO:COMMND"\GOTO 10070
10060   IF ERL=10050 AND ERR=305 THEN SAVE "MFO:COMMND" ELSE OFF ERROR
10070   END
```

Hints On Using Command Files

* Several **Command Files** can be kept on a disk and one made **active** by **assigning** it to **COMMND.SYS.** Assume we have the following command files stored on disk.

```
COMMND.ONE
COMMND.TWO
COMMND.TMP
```

We can make **COMMND.ONE** the **Active Command File** by using **FUP** as shown below.

EXAMPLE:

***MFO:COMMND.SYS=MFO:COMMND.ONE (CR)**

CAUTION: *This will delete whatever was previously stored as "COMMND.SYS."*

This creates a file **COMMND.SYS** on the disk using the contents of **COMMND.ONE.** Note that **COMMND.ONE** is still retained on the disk for backup purposes.

Similarly, to **deactivate** the Command File, just delete **COMMND.SYS** as shown below.

EXAMPLE:

***MFO:COMMND.SYS/D (CR)**

A useful application of this would be where **COMMND.SYS** on the disk is used to **download** the contents of the **Disk** to **E-Disk, assign** the E-Disk as the **system device, assign** a file from the disk (**COMMND.EDO:**) as the Commnd File for E-Disk, and then **run a user program.** The two files as they would appear on the disk are shown below.

```
*COMMND.SYS (CR)
T
F
EDO:/F
Y
EDO:=MFO:/W
EDO:/A
COMMND.SYS=COMMND.EDO:/R
/X
B
RUN "USER.PRG"
```

```
*COMMND.EDO: (CR)
T
B
RUN "USER.PRG"
```

***DISPLAYING THE CONTENTS OF THE COMMAND FILE WHILE IT IS ACTIVE**

Often it is useful to **display** the **instructions** contained in the **Command File** while it is active. This can be easily accomplished by adding an instruction to the Command File telling it to **list COMMND.SYS** while in **FUP.** This is illustrated in the sample Command File shown below.

EXAMPLE:

```
*COMMND.SYS (CR)
T
F
COMMND.SYS (Displays the File)
EDO:/F
Y
B
RUN"TEST 1.BAS"
```

***DISPLAYING MESSAGES WHILE THE COMMAND FILE IS ACTIVE**

If instead of displaying the contents of **COMMND.SYS** as was done above, we can display the contents of a different file. This allows us to **display a message** by putting it in the file. We will call the file **DSPLY.MSG.** It can be **generated** from **FUP** just as Command Files can.

EXAMPLE:

```
*DSPLY.MSG=KBO:
  (CR)
  (CR)
            PLEASE STANDBY!!!!
     DOWNLOADING FLOPPY TO E-DISK
  (CR)
  Z
```

It is possible to make use of the **attributes** capability of the 1720A when generating the **Display File** but that procedure will be covered in a later lesson.

To use the **Display File** merely place the instruction to display it in the command file as shown below.

EXAMPLE:

```
*COMMND.SYS (CR)
T
F
DSPLY.MSG (Displays Message)
EDO:/F
Y
EDO:=MFO:/W
EDO:/A
/X
B
RUN"USER.PRG"
```

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0097A-10U8009/SE EN

FLUKE®

Technical Data

17XXA Highlighted Learning Program B0098

Disk Initialization

FUP, the File Utility Program, is a file transfer and management program giving you flexible control over the files on floppy disks and on the optional Electronic Disk. FUP sets up a communications channel between devices and files recognized by the **Floppy Disk Operating System (FDOS).**

/F - FORMAT A DISK OR ELECTRONIC DISK

Before a floppy disk is used for the first time, its magnetic surface is completely blank. In this form it is not possible for the 1720A to write on the floppy disk until it has been **FORMATTED.** The formatting procedure writes addresses in the form of tracks and sectors onto the floppy disk so that specific locations may be referenced by the 1720A. FUP is used to format either the floppy disk or Electronic Disk.

SYNTAX: **[<DEVICE>], <DEVICE>]]/F<CR>**
DEFAULT: **<DEVICE>=SYO**

While multiple device formatting is possible we will deal with formatting one device.

This reduces the command line to:

```
[<DEVICE>] / F <CR>
```

If we wish to format a floppy disk we would insert a blank disk in the drive and use the following command line:

```
*MFO: / F <CR>
```

But if MFO: is the default device (MFO: is SYO:) the command line would be:

Example:

```
*/F <CR>
Really zero SYO:? YES <CR>
```

Notice that the 1720A asks for CONFIRMATION OF THE FORMATTING COMMAND. After **FORMATTING,** the **DEVICE DIRECTORY** will be **ZEROED.**

NOTE:** If a device containing **FILES** is **FORMATTED,** all **FILES** on the device are **ELIMINATED (erased).

Similarly, if MFO: is SYO: and we want to **FORMAT** the **ELECTRONIC DISK** we would use the following command line.

Example:

```
*EDO: F <CR>
Really zero EDO:? YES <CR>
```

/S - BAD BLOCK SEARCH

This command scans devices for **BAD BLOCKS.** It will print to the **OUTPUT FILE** the **NUMBER** of the **BAD BLOCK** and the **TOTAL NUMBER OF BAD BLOCKS** found. In addition, if any of the blocks have **SOFT ERRORS** (requiring more than one try to write or read valid information onto the block) it will indicate the **NUMBER** of **RETRIES** that were necessary.

SYNTAX: **[<OUTPUTFILE> =][<DEVICE>[,**
<DEVICE>]] / S <CR>

DEFAULT: <DEVICE> = SYO <OUTPUT FILE>
= KB0:

While it is possible to scan multiple devices for bad blocks we will deal with one device. This reduces the command line to:

```
[ <OUTPUTFILE> = ][ <DEVICE> ] / S <CR>
```

If we wish to scan a disk the command line would be:

```
*KBO: = MFO: / S<CR>
```

But since **KBO:** is a default condition and if **MFO:** is a default condition the command line reduces to **/ S** <CR>

Example:

```
*/ S <CR>

Block 273 has 1 entry
Block 275 has 1 retry
Block 326 has 3 retries
Block 351 is bad
Block 358 is bad

Total of 2 bad blocks found
```

Similarly if **EDO: is SYO:** we could scan a disk **(MFO:)** and print the diagnostics to a **FILE** on **EDO** called **SCAN.BLK.** The command line would be:

```
*EDO:SCAN.BLK=MFO: / S<CR>
```

But since EDO: is the default device the command line would reduce to:

```
*SCAN.BLK=MFO: / S<CR>
```

/L - List the DIRECTORY

This command prints the directory of the specified **DEVICE** to the specified **OUTPUTFILE.**

Syntax: **[<OUTPUTFILE>] = [<DEVICE>] /L <CR>**

Default: <**OUTPUTFILE**> = **KBO** (console device)
<**DEVICE**> = **SYO** (system device)

If we wish to list the directory of a floppy disk on the display **(KBO:)**, the command would be:

```
*KBO: = MFO: /L <CR>
```

but since **KBO:** is the default condition and if **MFO:** is the default condition, the command simplifies to **/L** <CR>

Example:

```
*/L <CR>

 Directory of SYO: on 3-Oct-79 at 16:34

Name.Ext          Size     Date

FDOS.SYS          13       25-Sep-79
CONMON.SYS        2        20-Sep-79
TIME.CIL          2        20-Sep-79
SET.CIL           1        20-Sep-79
FUP.CIL           9        20-Sep-79
BASIC.CIL         50       20-Sep-79
EDIT.CIL          9        20-Sep-79
SELECT.BAS        12       20-Sep-79
DEMO.BAS          46       21-Sep-79
LIST.13E          5        25-Sep-79
IEEE.BAS          1        2-Oct-79
ENTER.BAS         4        3-Oct-79
8520.1            29       3-Oct-79
8520.BAS          29       3-Oct-79

Total of 212 blocks in 14 files, 186 free blocks
```

Similarly, if we want to list the above directory out to a printer connected to the RS-232 port **KB1:,** the following command would be used:

```
*KB1: =MFO:/L <CR>
```

And due to the default conditions it would simplify to:

```
*KB1: =/L <CR>
```

Finally, if **SYO:** is **MFO:** and we wish to list on the display the directory contained on electronic disk, the command would be:

```
*KBO:=EDO:/L <CR>
```

And due to the default conditions we would use:

```
*EDO:/L </CR>
```

/E - EXTENDED DIRECTORY listing

Since the 1720A employs a **CONTIGUOUS file** structure, there can be blocks of **UNUSED ENTRIES.** These won't be displayed with the **/L** command. If a listing of these empty entries is desired, however, the **/E** command can be used instead of the **/L** command.

Example:

```
*/E <CR>

Directory of SYO: or 3-Oct-79 at 16:34

NAME.EXT          SIZE     DATE

FDOS.SYS          13       25-Sep-79
CONMON.SYS        2        20-Sep-79
TIME.CIL          2        20-Sep-79
SET.CIL           1        20-Sep-79
FUP.CIL           9        20-Sep-79
BASIC.CIL         50       20-Sep-79
<NOT USED>        30
DEMO.BAS          46       21-Sep-79
<NOT USED>        5
ENTER.BAS         4        3-Oct-79
<NOT USED>        236

Total of 127 blocks in 8 files, 271 free blocks
```

FLUKE®

The angle brackets indicate blocks of **UNUSED ENTRIES** located between files. The **/E** command then indicates not only the amount of unused blocks but also where they are distributed on the disk.

/P - PACK A DEVICE

In order to best utilize the unused blocks on a disk it is often desirous to have all the free space on a disk in one **Contiguous Section** located after the existing files. This is accomplished with the **/P** command.

SYNTAX: [<**DEVICE**> [, <DEVICE>]] **/P** <**CR**>

DEFAULT: <**DEVICE**> = **SYO**

While multiple device packing is possible we will deal with packing one device. This reduces the command line to:

[<DEVICE>] /P <CR>

Here's how we take the disk contents from the previous **/E DIRECTORY LISTING** and pack them. The command line would be:

***MFO:/P <CR>**

But if **MFO:** is the default device the command line reduces to **/P <CR>.**

Example:

***/P <CR>**

***/E <CR>**

Directory of SYO: on 3-Oct-79 at 16:40

NAME.EXT	SIZE	DATE
FDOS.SYS	13	25-Sep-79
CONMON.SYS	2	20-Sep-79
TIME.CIL	2	20-Sep-79
SET.CIL	1	20-Sep-79
FUP.CIL	9	20-Sep-79
BASIC.CIL	56	20-Sep-79
DEMO.BAS	46	21-Sep-79
ENTER.BAS	4	3-Oct-79
<NOT USED>	271	3-Oct-79

Total of 127 blocks in 8 files, 271 free blocks.

Notice that now the all free space on the disk is located in one section and there are no unused entries between files.

If we want to pack the **ELECTRONIC DISK** when **SYO:** is the floppy disk (**MFO:**) we would use the following command line:

***EDO:/P <CR>**

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.
B0098B-10U8208/SE EN

FLUKE®

Technical Data

17XX Highlighted Learning Program B0099

Modification of Files

FUP, the File Utility Program, is a file transfer and management program giving you flexible control over the files on floppy disks and on the optional Electronic Disk. FUP sets up a communications channel between devices and files recognized by the **Fluke Disk Operating System (FDOS).**

FUP can be used to easily accomplish the following file modifications:

/R RENAME A FILE

SYNTAX: <DESTINATION FILE> = <SOURCE FILE> /R <CR>

Default: **None**

The **INPUT FILE** is **RENAMED** according to the name as specified in the **OUTPUT FILE** list.

Example:

To take the file **193A.BAS** and **RENAME** it to **1953A.BAS** we would use the following command line:

```
*1953A = 193A/R <CR>
```

NOTE: *If the devices don't correspond, an error message is given. If the output file already exists, an error message is given.*

/D DELETE A FILE

This command **DELETES** the specified file or files.

SYNTAX: **<FILE NAME> [, <FILE NAME>] /D <CR>**

DEFAULT: None

Example:

If it is desired to **DELETE** the files **METER.BAS** and **TEST.488** the following command line would be used:

```
*METER, TEST.488/D <CR>
```

NOTE: *Limited to eight files at a time.*

/M MERGING ASCII FILES

SYNTAX: [<DESTINATION FILE>=]<SOURCE FILE>[, <SOURCE FILE>]/M <CR>

DEFAULT: **<DESTINATION FILE> = KBO:** (CONSOLE DEVICE)

CAUTION: *Line numbers of the files to be merged should not overlap.*

Example:

To MERGE the content of **OLD.BAS** and **NEW.BAS** and create a new file **NEW.BAS** all on the system device we would use the following command line:

```
*NEW = OLD, NEW/M <CR>
```

NOTE: *There can be a maximum of **8 FILES** specified in the **SOURCE** (or right side of the command).*

ERROR FREE FUP

FUP is a powerful tool. Good programming practices should be employed to avoid pitfalls when using it.

1. Make a back up copy of your disk before modifying or packing any files. If you do make an error afterwards, it won't be catastrophic.
2. Verify the results of each FUP operation you do. View the directory or files as necessary to accomplish this. It is better to discover if you have inadvertently erased a file or disk at the finish of each operation than when the program can't be recovered.
3. Verify your FUP commands as typed before you execute them. Imagine what would happen if you typed MFO:/Z when you really meant to type MMO:/Z.
4. Adequately identify your Floppy disks by writing something meaningful on the label with a felt tip pen. Formatting an already recorded disk which has a blank label can ruin your day. Better yet, do a /L command on a "blank" disk before you format it. If you get a directory listing, you have saved a disk.
5. Routinely Pack and Search your disks for bad blocks. Disks which have bad blocks should have their files copied onto another disk. If re-formatting the questionable disk doesn't get rid of the bad blocks, the disk should be discarded.
6. Be sure to use /B or /W when transferring binary files. /B will handle any ASCII files as well.
7. Be sure to store your system command file under a temporary name before creating a new command file.

ERROR MESSAGES

Listed below are the error messages and their explanations which can be encountered in FUP.

DEVICE ERROR
This error indicates a fatal error on a I/O device. Examples are disk CRC errors, magnetic tape parity errors.

?DEVICE NOT READY
This means that the accessed device is not ready. Remedies are loading a diskette or closing the disk door.

?WRITE PROTECTED
The diskette has a write protect tab. This can be fixed by either removing the tab or using a non protected diskette.

?NOT A VALID DEVICE
This means that a device has been specified for which there is no support in FDOS. Misspelling is often the cause.

?FILE NOT FOUND
This indicates that the required file could not be found on the specified device.

?NO ROOM FOR USER ON DEVICE
This indicates that the created file caused by either merging or copying exceeds the amount of available space.

?NO END-OF-FILE
An ASCII file had been used as an input file, while not containing an End-Of-File mark.

?DEVICES DO NOT MATCH
A rename option with different devices has been given.

?NOT A FILE STRUCTURED DEVICE
Certain operations like packing or listing a directory are legal only on file structured devices.

?SYNTAX ERROR
A command line as input by the user did not meet the syntax specification as required for that particular command.

?TOO MANY FILES
More than eight files had been specified either in the input or output file specification.

?ILLEGAL OPTION
An unrecognized option was selected. Probably a typing error.

?NOT A VALID FILE NAME
A filename with illegal or too many characters had been entered.

USING FUP TO DE-BUG BASIC PROGRAMS

Files which have been OPENED for the storage of ASCII data can be viewed via FUP by typing the file name followed by the carriage return. If it is a long file the paging keys should be used. Data is stored on the disk by the PRINT command in identically the same format that is used to PRINT via a mechanical printer.

The data being stored on the disk can be considered as a printed page. In this context, errors such as "input line too long", "too much data typed" and "not enough data typed" become meaningful.

If your basic program is having problems outputing data to the disk or inputing data from the disk, viewing the data file on the CRT via FUP can provide significant insight.

FLUKE ®

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0099B-10U8208/SE EN

17XXA Highlighted Learning Program B0100

Copying (Transferring) of Files

The File Utility Program is the ideal program to use for duplication of files and disks. FUP inputs information directly from the original disk to the output device **For Temporary Storage** for writing on the disk you are duplicating. More than one file can be duplicated at a time; in fact, it is possible to duplicate an entire disk with one command, providing the original does not exceed 89 blocks when MMO: is the destination or 254 blocks when EDO: is the destination 508 blocks if 2 E-Disks are used).

ZERO THE DIRECTORY

There are two reasons for zeroing a directory. In the first case, you have a floppy disk or E-Disk you wish to erase. In the second case, you are using **EDO:** or **MMO:** as **TEMPORARY STORAGE** for duplicating floppy disks (floppy disk #1 to temporary storage; temporary storage to floppy disk #2) and the original floppy disk contains more blocks than the temporary storage. Consequently, the temporary storage must be copied to more than once and its directory must be zeroed before it can be copied to again.

When the /**Z** command is used, the **DIRECTORY** of the specified **DEVICE** is **ZEROED OUT**, that is, **ALL FILE HEADERS WILL BE WIPED OUT.** (Note that the file itself still exists, just the pointer, or file header is erased.)

SYNTAX: = **[**<**DEVICE**> [, <DEVICE>]]/**Z** <**CR**>

DEFAULT: <**DEVICE**> = **SYO**

While multiple device zeroing is possible we will deal with zeroing one device. This reduces the command line to:

```
= [<DEVICE> ]/Z <CR>
```

If we wish to zero the directory on a disk we would use the following command line:

```
*MFO:/Z <CR>
```

But if MFO: is the default device the command line would be:

```
*/Z <CR>
Really zero SYO.? YES <CR>
```

Notice that the 1720A asks for **CONFIRMATION** of the **ZEROING COMMAND.**

Similarly if **MFO:** is **SYO:** and we want to **ZERO** the **DIRECTORY** on **ELECTRONIC DISK** we would use the following command line:

```
*EDO:/Z <CR>
Really zero EDO: ? YES<CR>
*
```

If a device contains a **DIRECTORY** it had to have been previously **FORMATTED.** Therefore after **ZEROING** an **EXISTING DIRECTORY,** the device **DOES NOT** require **FORMATTING** in order to use it again.

/W - WHOLE COPY A DEVICE

The /**W** command allows **FAST DUPLICATION OF DISKS.**

All files from the **SOURCE DEVICE,** starting at the specified **START FILE,** are **COPIED** to the **DESTINATION DEVICE** until no more exist or the destination device is full.

SYNTAX: [<**DESTINATION DEVICE**>] = **[SOURCE DEVICE**> [<START FILE>]]/**W** <**CR**>

DEFAULT: <**DESTINATION DEVICE**> : <**SOURCE DEVICE**> = **SYO**

If **NO START FILE** is specified, the **WHOLE DEVICE** is copied. The copy process continues until there is **NO ROOM** on the **OUTPUT DEVICE** for another file or until **ALL FILES** have been **COPIED.** If a **CTRLC** is given during a file transfer, the copy process is **ABORTED** when copying of the **CURRENT FILE** has been completed.

When a **CRTL P** is given, the copy process is **ABORTED** and the **CURRENT FILE** will be **CLOSED** even though the **WHOLE FILE MIGHT NOT** have been **TRANSFERRED.** The names of all the copied files are displayed on the console terminal.

If we wanted to **COPY** the contents of a **DISK** to **ELECTRONIC DISK** we would use the following command line:

```
*EDO: = MFO:/W <CR>
```

But if **MFO:** is the default device the command line would be:

```
*EDO: = /W <CR>

Copying   FDOS.SYS
Copying   CONMON.SYS
Copying   TIME.CIL
Copying   SET.CIL
Copying   FUP.CIL
Copying   BASIC.CIL

*
```

When the * is displayed, it indicates the transfer is completed and the contents of **MFO:** are now in **EDO:**

If we wished to make a **DUPLICATE COPY** of the disk we have just transferred, we merely have to insert another **FORMATTED** disk in the drive and transfer the **FILES** back from **EDO:**. The command line would be:

MFO: = EDO:/W <CR>

But since **MFO:** is the default device the command line would be:

```
*= EDO:/W <CR>

Copying   FDOS.SYS
Copying   CONMON.SYS
Copying   TIME.CIL
Copying   SET.CIL
Copying   FUP.CIL
Copying   BASIC.CIL

*
```

When the * is displayed it indicates the transfer is completed and the contents of **EDO:** are now on the **NEW DISK** which had been inserted in drive **(MFO:).**

Since a disk can store **200K BYTES** of information and one **ELECTRONIC DISK** can store **128K BYTES** it would require **2 PASSES** to duplicate a disk with more than **128K BYTES (DUAL ELECTRONIC DISKS would eliminate this problem as there would be 256K BYTES** available).

We will assume that the disk files start with **FDOS.SYS,** and the **SECOND PASS** will copy the remaining files starting with the one after **8520A.BAS** which we will assume to be **NEW.BAS** and end with the last file **TEST.BAS.**

The procedure is as follows: (Assume **MFO: = SYO:**)

FIRST PASS

Insert **DISK #1** (the disk to be copied) in the drive.

```
* EDO: = /W <CR>

Copying FDOS.SYS
     ↓      ↓
Copying 8520A.BAS

No room for user on device

*
```

Notice that the 1720A indicates when there is no more room on the device to copy any additional files. Remove **DISK #1** and insert **DISK #2** (the duplicate disk) in the drive.

```
* = EDO:/W <CR>

Copying FDOS.SYS
     ↓      ↓
Copying 8520A.BAS

*
```

SECOND PASS

Since there is no more room on **EDO:** it is necessary to **ZERO** its **DIRECTORY** before transferring the remaining files to it.

```
* EDO:/Z <CR>

Really Zero: ? YES<CR>

*
```

Now remove **DISK #2** and insert **DISK #1** in the drive.

Since we have already partially copied the files on **DISK #1** it is now necessary to **SPECIFY** a **START FILE (NEW.BAS)** in order to transfer the remaining files to **EDO:**.

```
* EDO: = NEW.BAS/W <CR>

Copying NEW.BAS
     ↓      ↓
Copying TEST.BAS

*
```

At this point all the remaining files have been transferred to **EDO:**.

Remove **DISK #1** and insert **DISK #2** in the drive.

```
* = EDO:/W <CR>
Copying NEW.BAS

Copying TEST.BAS
*
```

At this point, all the files on **DISK #1** have been copied onto **DISK #2.**

COPY OF FILES USING MAIN MEMORY (MMO:)

If a 1720A is not equipped with **ELECTRONIC DISK** it is still possible to copy files as was previously done with **EDO:** by using the **MAIN MEMORY.**

FUP can assign all the available **MAIN MEMORY** to act as a **MASS STORAGE DEVICE.** A special device will be created by **FUP** to accommodate this function. This device (**MMO:**) is only valid in **FUP** and is main memory. Its size depends on the amount of available main memory. Using the **TRANSFER** or **WHOLE COPY** commands, the user can fill this device with files from the disk, change the diskette and copy form **MMO:** to the disk. (See **TRANSFERRING BINARY FILES** and **TRANSFERRING ASCII FILES** on the following pages.) If **MMO:** can not hold all the files, the user must **PERFORM ADDITIONAL PASSES** in order to copy the remainder of the files. Note that **MMO:** will be automatically zeroed when **FUP** is started.

CAUTION: Certain operations like formatting a floppy disk will cause MMO: to be zeroed.

Example:

```
* MMO: = /W
Copying FDOS.SYS
Copying CONMON.SYS
Copying BASIC.CIL
*
```

NOTE:MMO: does **not** require formatting.

At this point all files have been transferred and the user must now swap disks.

```
* = MMO:/W
Copying FDOS.SYS
Copying CONMON.SYS
Copying BASIC.CIL
*
```

REMEMBER, as when using **EDO:** and making **MULTIPLE PASSES** it is necessary to **ZERO THE DIRECTORY** on **MMO:** and **SPECIFY A START FILE** after the **FIRST PASS.**

LIST ASCII FILES

SYNTAX: [**<DESTINATION FILE> = <SOURCE FILE> <CR>**

DEFAULT: **<DESTINATION DEVICE>=SYO:** if a **<FILENAME>** is specified

<DESTINATION FILE> = KBO: if **<DESTINATION FILE>** is omitted.

If **NO DESTINATION FILE** is **SPECIFIED** the **SOURCE FILE(S)** will be printed on the Console Terminal. This gives us a convenient way to list **ASCII** files on the Console Terminal.

Example:

```
*METER.488 <CR>
```

will cause the file to be listed on the Console Terminal.

Example:

```
* KB1: = METER.488 <CR>
```

will cause the file to list out to a printer connected to RS-232 port KB1:

TRANSFERRING BINARY FILES

In order to transfer a BINARY FILE it is necessary to use the **/B** command.

For instance if **MFO:** = **SYO:** and we wish to transfer the file **BASIC.CIL** over to **ELECTRONIC DISK** we would use the following command line:

```
* EDO:= BASIC.CIL/B <CR>
```

***NOTE:** Binary transfers can be made for all types of files, (ASCII, etc.).*

TRANSFERRING ASCII FILES

If **MFO:** = **SYO:** and we wish to transfer the files **METER.BAS** and **NEW.TST** from the disk over to **ELECTRONIC DISK** we would use the following line:

```
* EDO:= METER, NEW.TST <CR>
```

/T TURN OFF ERROR CHECKING AND TRANSFER/COPY AN ASCII FILE

SYNTAX: Identical to copy command

On those rare occasions when one or more blocks in a file have gone bad, the user may want to recover as much of the file as possible. They can do so by defeating some error checks. (Device error and No End-of-file) If a device error occurs the block will be transferred as is, that is, it may contain garbage. If an end-of-file mark has disappeared a new one will be automatically appended.

Example:

```
* EDO: = METER.488/T <CR>
```

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.
B0100B-10U8208/SE EN

FLUKE®

Technical Data

1720A Highlighted Learning Program B0123

E-DISK ™

The optional electronic disk is a high-speed, file-structured mass storage system. It works just like floppy disk, differing only in capacity and speed. E-Disk files can easily be transferred to a floppy disk for permanent storage. When installed, the E-Disk occupies one or both of the circuit module slots marked "SPARE".

FEATURES

- High speed
- Large capacity
- Works just like the floppy disk
- Battery backup available for power interruptions

HIGH SPEED

The speed of a file storage system is normally described in two ways:

1. Access time: The time lag between identifying a file request and the beginning of file transfer.
2. Transfer rate: The rate that data is transferred after it has been accessed.

Table 1 compares the hardware speed performance of the E-Disk with that of the floppy disk. Note that software processing overhead is in addition to these times.

	E-DISK	FLOPPY DISK	E-DISK ADVANTAGE
ACCESS TIME (milliseconds)	.018	550 (average)	30,600:1
TRANSFER RATE (bytes/second)	133,120	15,625	8.5:1
TIME TO FILL MAIN MEMORY (seconds)	0.46	3.9	8.5:1

Among the benefits of this high speed are:

* Most lexically-saved forms of BASIC programs load so quickly that the short delay time for loading is not apparent to an operator.

* Sequential files and random access virtual arrays can be built up and accessed within a running program without significantly slowing it down.

LARGE CAPACITY

Programs and data files on the E-Disk are organized as a sequence of 256 or 512 data blocks. Each data block contains 512 bytes of information. The first two blocks (1K bytes) are reserved for a file directory.

A second E-Disk module doubles storage capacity without the need to address each module separately. Because of this, a single large file can occupy two modules.

The largest file that a single module E-Disk can store is 254 blocks (127K bytes). A dual module system can store 510 blocks (255K bytes). The maximum number of files is 72.

Some of the benefits of the large capacity of an E-Disk are:

* Sequential files and random access virtual arrays can be much larger than would be possible in main memory without significant sacrifice in processing speed.

* Large programs that will not fit into main memory can be structured into modules, and chained together. Although this can be done with the floppy disk, use of the E-Disk nearly eliminates the delays of loading each program module into memory, and frees the floppy disk for collection of processed data.

* Floppy disk files that are too large to fit in main memory can be duplicated. (A dual-module system is required for files larges than 127K bytes.)

WORKS JUST LIKE THE FLOPPY DISK

Techniques for using the E-Disk are identical to the floppy disk. Like any disk, the E-Disk must be formatted before use, and contains a file directory.

The E-Disk is referred to as EDØ: in program statements and utility commands.

When installed, the E-Disk can easily be set up to be the default System Device for files not given a named location.

Some of the benefits of working like the floppy disk are:

* Programming is simplified. Taking advantage of the E-Disk requires no new techniques. The only difference is referring to EDØ: (the E-Disk) in place of MFØ: (the floppy disk).

* By making use of the System Device concept, programs can be structured to run on any 1720A system, taking advantage of the E-Disk whenever it is installed. In such a case, an E-Disk will increase the speed of processing sequential and virtual array files, and eliminate some requirements for an operator to exchange disks.

E-Disk ™ is a trademark of the John Fluke Mfg. Co., Inc.

FLUKE®

BATTERY BACKUP

The E-Disk will hold its contents intact when power is removed, provided the E-DISK BATTERY switch on the rear power supply panel is set to ENABLE, and there is an internal or external source of battery power. A fully-charged internal battery will support a single-module E-Disk for about one hour, or a dual-module E-Disk for about 30 minutes. An external battery can easily be connected through the rear REMOTE INTERFACE connector to extend this time indefinitely.

If you do not plan to make use of the battery backup feature for the E-Disk, battery life can be extended by leaving the feature disabled. This will prevent the battery from having to discharge and recharge regularly.

SOME EXAMPLES

1. This Immediate Mode BASIC command will store the lexical form of the program currently in main memory on the E-Disk, in a file named "DEMO.BAL":

```
SAVEL "ED0: DEMO"
```

2. This FUP command copies a file from the floppy disk to the E-Disk:

```
ED0:DATA.T9=MF0:DATA.T9
```

3. The following FUP command is identical to the previous one, if the E-Disk is the default System Device. The file name is carried over unchanged when it is not specified:

```
=MF0:DATA.T9
```

4. If a different floppy disk is then inserted, the following FUP command writes a duplicate copy of the file onto the second disk from the copy on the E-Disk. This also assumes the E-Disk is the System Device:

```
MF0:=DATA.T9
```

5. When a program uses a data file on the E-Disk, a channel must first be opened. Input and output statements then refer to the channel number. The following Fluke BASIC sequence opens an existing file on the E-Disk for sequential input, and then reads its contents into the string M3$:

```
2400 OPEN "ED0:DATA.T8" AS FILE 4
2410 INPUT #4, M3$
```

6. The following Fluke BASIC sequence opens a new file on the default System Device for sequential output, and then stores a message in it. When the E-Disk is installed, this file will normally go onto it unless the System Device was reassigned:

```
3920 OPEN "MSG4" AS NEW FILE 2 SIZE 1
3930 PRINT #2, "End of Frequency Test"
```

7. A random access virtual array requires a dimension statement referencing a previously opened channel. The following sequence opens a random access (DIM) file on the E-Disk, dimensions an integer array into it, and then places a value stored in integer variable N% into one of the array elements. Note that virtual array channels are bidirectional. The NEW specification causes the array elements to be initially set to zero, replacing any existing file found with the same name. Virtual arrays are discussed in a separate HLP bulletin.

```
1020 OPEN "ED0: DATA.43" AS NEW DIM FILE
     3 SIZE 1
1030 DIM #3, D% (15,31)            !512 elements
.
.
.
4960 D% (4,6) = N%                 !N% Previously defined
```

E-DISK TERMINOLOGY

Access Time
The time lag between identifying a file request and the beginning of file transfer.

ASCII
The American Standard Code for Information Interchange is a set of defined 7-bit binary code patterns representing the full alphabet, numbers, and many useful symbols and control characters.

Bit
A single unit of program or data information, set to either one or zero. Bit is a contraction of binary digit.

Block
A fixed size of data selected to be convenient for storage or transfer operations. In the 1720A Instrument Controller, a block is 512 bytes.

Block-Structured
A block-structured device transfers and stores data one block at a time, accumulating smaller amounts in a temporary buffer until a block is available. E-Disk is a block-structured device.

Buffer
A temporary memory storage area for accumulating information until enough is available for the next operation. E-Disk transfers use a 1-block (512-byte) buffer.

Byte
Eight data bits set to any pattern of ones and zeros with defined meaning, such as binary numbers or the ASCII code.

Chaining
The technique of separating a program into task-oriented modules. Each module contains a RUN "next module" statement linking it to the next task. Variables can be passed to the next program module through a common storage area, and data in virtual arrays can be left open for subsequent use. Only the module currently in use occupies space in main memory.

E-Disk ™
The optional electronic disk is a mass file-storage system constructed of high-speed, solid-state memory and designed to function as a serial transfer block-structured device. With appropriate operating system software, it appears to the system as a file-structured device.

File
A collection of information designated by name as a unit.

File-Structured
A file-structured device transfers and stores information by file units.

Formatting
Formatting is a process of preparing a file-storage device to accept files. After verifying the integrity of each block, an identifying and timing pattern is recorded throughout. The 1720A Instrument Controller accomplishes this with the /F command option of the FUP Utility Program.

K Byte
1024 bytes.

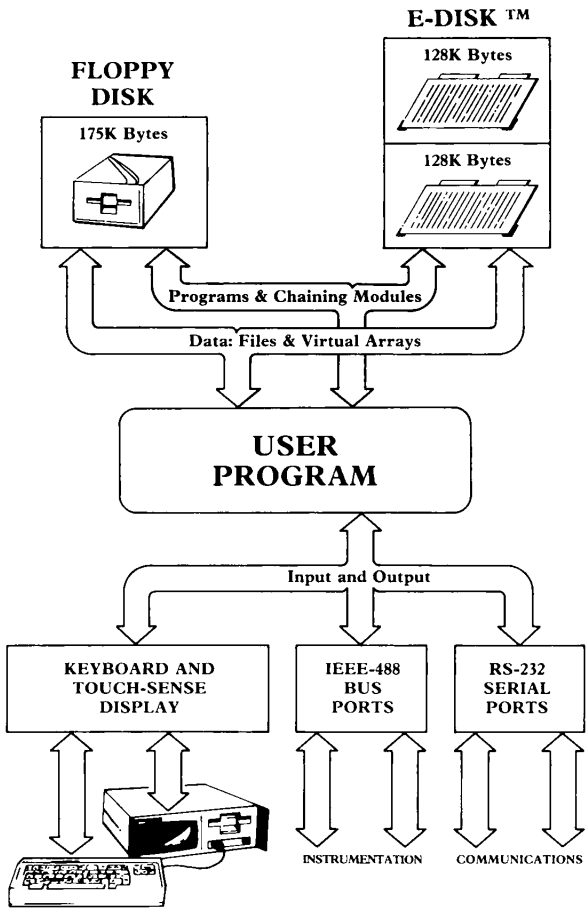
Lexical File
A BASIC language program that has been processed by the interpreter into the form that is used in main memory. In this form, the program file occupies less space and loads into memory quicker. Line numbers, keywords, operators, and branch pointers are processed into binary form. Lexical files are compatible only with the version of the BASIC interpreter that generated them.

Random Access File
The contents of a random access file, such as a virtual array, can be accessed in any desired sequence by referencing individual elements in the file.

Sequential Access File
The contents of a sequential access file can only be accessed from start to end.

Transfer Rate
The rate that a file is transferred after it has been accessed.

Virtual Array
A collection of data defined by a DIMension statement that references an open channel to a file-structured device, such as the E-Disk. The data array is available to the user program just as if it were present in main memory. For this reason it is called virtual.



E-DISK ™ FUNCTIONS

John Fluke Mfg. Co., Inc.
P.O. Box 43210, Mountlake Terrace, WA 98043
800-426-0361 (toll free) in most of U.S.A.
206-774-2481 from AK, HI, WA and Canada
206-774-2398 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.
B0123A-10U8011/SE EN

Technical Data

1720A Highlighted Learning Program B0142

Basic Main Memory Arrays

Array variables include Reals (floating point), Integers and Strings. Interger array variable names are followed by %; string array variable names are followed by $.

Array variable names may be one or two characters (first character may be any upper case letter; second character may be any upper case letter or any number 0 through 9 or may be omitted); note these are the same names as are used with simple variables. Note also that IF, ON, OR, TO, FN, LN, PI and AS are not legal variable names.

There are only 954 possible names for each type of simple variable in FLUKE BASIC. This may seem to be adequate, however, it is not enough to store the readings from a typical digital voltmeter measurement. Moreover, the possibilities for meaningful names from the 954 possible names are very limited. Consider also, the fact that 954 BASIC statements would be required to assign a value to each possible simple variable name. For these reasons, an alternative to simple variables is necessary. The need for an array of variables (each array variable is called an element) that can be identified by a number as part of the name, is evident.

Suppose you had 10 numbers you needed to assign to simple interger variables:

```
N0% = 26792   \   N5% = 26797
N1% = 26793   \   N6% = 26798
N2% = 26794   \   N7% = 26799
N3% = 26795   \   N8% = 26800
N4% = 26796   \   N9% = 26801
```

You could do it as shown above or you could dimension an integer array of 10 elements:

DIM N%(9%)   dimensions N%(0%) through N%(9%)

Each of the 10 array elements is a variable and is capable of storing data like any simple variable. In other words, DIM N%(9%) creates an additional 10 variables for the program to use.

Assignments can be made to the array element variables in the same manner as you did with the simple variables:

```
N%(0%) = 26792   \   N%(5%) = 26797
N%(1%) = 26793   \   N%(6%) = 26798
N%(2%) = 26794   \   N%(7%) = 26799
N%(3%) = 26795   \   N%(8%) = 26800
N%(4%) = 26796   \   N%(9%) = 26801
```

So far, the only difference between the simple variable assignments and the array element assignments, are the addition of brackets around the numbers, and the prerequisite DIM statement to reserve main memory space for the array variables and to establish the array element names as valid variable names in the program.

The numbers in the names of the array element variables serve the same purpose as the numbers in the simple variables, i.e., they are part of the variable name and make the named variable unique and distinguishable from the other variables. N1% is not the same variable as N%(1); both are allowed. The number within the brackets of the array element variable has one important additional capability: it can be a variable. Bracketed element numbers are usually called subscripts.

Now you can make the same assignments in a more flexible and structured manner using a FOR/NEXT loop:

```
10  DIM N%(9%)
20  FOR I% = 0% TO 9%
30  READ N%(I%)
40  NEXT I%
50  DATA 26792, 26793, 26794, 26795, 26796
60  DATA 26797, 26798, 26799, 26800, 26801
70  END
```

Enter and RUN the above program; then from the Immediate Mode, type:

PRINT N%(3)   <CR>

The display should indicate "26795" which is the 4th value in the DATA statement. The other 9 elements can be verified in like manner, or you could simply type:

PRINT N%(0..9)   <CR>

which is the equivalent of:

```
FOR I = 0 to 9
PRINT N%(I)
NEXT I
```

The above statement functions like a FOR/NEXT loop and increments the array element subscript numbers from 0 through 9; the result is to PRINT the entire array of variable elements with a <CR> <LF> after each element. To suppress the <CR><LF> type:

PRINT N%(0..9); <CR>

This notation can be used for any contiguous list of elements, e.g.,:

PRINT N%(3..7)

FLUKE®

Now you have an even simpler method of making assignments:

```
10  DIM N%(9%)
20  READ N%(0..9)
30  DATA 26792, 26793, 26794, 26795, 26796
40  DATA 26797, 26798, 26799, 26800, 26801
50  END
```

Until now you have been working with a one-dimensional array. It may help you to think of a one-dimensional array as a one-row matrix. The above example represented as a matrix would be:

	← Columns 0 through 9 →									
ROW	N%(0)	N%(1)	N%(2)	N%(3)	N%(4)	N%(5)	N%(6)	N%(7)	N%(8)	N%(9)

FLUKE BASIC supports String Arrays and REAL (floating point) Arrays as well as Integer Arrays. Dimensions and Subscripts must be ≤32767 (the maximum integer size). Subscripts may appear as mathematical expressions, e.g.,:

```
V$(A% + 3)="Volts"                R(13/N)=1.1412
SN$(SQR(A%))="Serial No."         DIM N%(X% + 3%)
```

If a subscript is not an integer, BASIC rounds the subscript to an integer.

Two Dimensional (Double Subscripted) Arrays

An array element with two subscripts can be thought of as a variable with another character in its name. This additional subscript gives the program the ability to create more elements than is possible with only one subscript. The additional subscript also gives the program the capability of building a matrix with ROWs as well as COLUMNs.

Suppose you had 3 production shifts and you wanted to store the total number of instruments produced by each shift for one week (5 days). The following program is an example of one way to do this task:

```
10   DIM PS%(2%,4%)          ! Dimension a 3x5 array
20   READ PS%(0..2,0..4)     ! READ 15 values
30   !   SHIFT  DAY  DAY  DAY  DAY  DAY
40   !     1     1    2    3    4    5
50       DATA   10,  12,   9,   7,   10
60   !     2
70       DATA    9,  14,  10,  11,   11
80   !     3
90       DATA   11,   7,   6,   8,   10
100  FOR I% = 0% TO 2%
110  PRINT PS% (I%,0..4);
120  PRINT
130  NEXT I%
140  END
```

The previous program takes the data and PRINTs it in the same arrangement as the DATA statements. Note the subscript order is always (ROW, COLUMN). The matrix for this array is shown below:

		← PS$(ROW NO., 0..4) →				
		Col. 1	Col. 2	Col. 3	Col. 4	Col. 5
PS%	ROW 0	PS%(0,0)	PS%(0,1)	PS%(0,2)	PS%(0,3)	PS%(0,4)
(0..2,	ROW 1	PS%(1,0)	PS%(1,1)	PS%(1,2)	PS%(1,3)	PS%(1,4)
Col. No.)	ROW 2	PS%(2,0)	PS%(2,1)	PS%(2,2)	PS%(2,3)	PS%(2,4)

Multiple Arrays

More than one array may be dimensioned in a program, e.g.,:

```
10  DIM A%(3,5), B%(6,10), A$(10,100)
20  DIM RL(1000)
30  DIM A5(10,10)
:
:
```

Redimensioning Not Allowed

BASIC programs are allowed to execute a DIM statement for each variable only once. An error will result if the program attempts to execute a specific DIM statement more than once. For this reason, DIM statements should appear early in the program and should not be included in subroutines. The only way around this is to re-RUN the program. RUN causes BASIC to forget all previously executed DIM statements.

Serial Storage of Arrays on the Disk

The following examples illustrate how array data can be serially stored and retrieved from the disk. You may use a different array name to retrieve data than you did to store the data; if the arrays are alike in type (integer, real or string) and dimensions. Note "NEW" in line 110 indicates a new file is being created on the disk.

Data Storage:

```
10  !"EXAM1.BAS"
100 CLOSE 1                            ! INITIALIZE
110 OPEN "EXAM1.DAT" AS NEW FILE 1 SIZE 1   ! RESERVE AND NAME DISK SPACE
120 DIM A$(5%)
130 FOR I%=0% TO 5%
140    A$(I%)=CHR$(65%+I%)           ! ASSIGN LETTERS "A" THROUGH "F"
150 NEXT I%
160 PRINT #1, A$(0%..5%)             ! STORE ON DISK
170 CLOSE 1                          ! CLOSE FILE
180 END
```

Data Retrieval:

```
10  !"EXAM3.BAS"
100 CLOSE 1                            ! INITIALIZE
110 OPEN "EXAM1.DAT" AS OLD FILE 1
120 DIM A$(5%)
130 !
140 INPUT LINE #1, A$(0%..5%) ! INPUT FILE DATA
150 !
160 CLOSE 1                  ! CLOSE FILE
170 PRINT A$(0%..5%)         ! DISPLAY DATA
180 END
```

Note "OLD" in line 110 indicates this file already exists on the disk.

Disk Storage Size Requirements

Before a Main Memory Array can be stored, space must be reserved for it on the disk. When a new file is OPENed, the largest available contiguous space on the storage medium (floppy disk or E-disk) is allocated for the single file, unless the SIZE is included in the OPEN statement. If two NEW files are OPENed without a SIZE statement and there is only one contiguous space available on the disk, the Operating System will display "? I/O error 306..." telling you there is no more room on the storage device, when the attempt is made to OPEN the second file. This will happen even though there is enough room on the disk for all the data you plan to store in each of the files.

Disk Size Calculation

SIZE must be stated as an integer number of BLOCKS (1 BLOCK = 512 BYTES). An Array file may contain more than one array. File SIZE must be large enough to equal or exceed the total number of storage bytes required by all of the REAL elements, INTEGER elements and STRING elements you plan to use in the Array file.

Array values are stored on the disk as ASCII values. One byte of disk storage is required for each character stored.

Disk requirements for serial storage (no comma after the variable):

1 byte per significant digit or string character
1 byte for the sign (even though it may be + and not be displayed)
1 byte for decimal point (reals only)
1 byte for an included space (except with PRINT USING) for reals and integers
2 bytes for <CR> <LF>
1 byte for EOF (end of file) character

EXAMPLES

PRINT #1, 3%	requires 5 bytes
PRINT #1, -3%	requires 5 bytes
PRINT #1, USING "S#", -3%	requires 4 bytes
PRINT #1, 3.285	requires 9 bytes
PRINT #1, -3.285	requires 9 bytes
PRINT #1, USING "S#.###", -3.285	requires 8 bytes
PRINT #1, "12345"	requires 7 bytes

Example for Disk Size Calculation

10 DIM S$(100), I%(100), R(100)

Assumptions:

1. All string elements = length of 10 characters.
2. PRINT USING is utilized to insure all reals are same length, and all integers are the same length.

```
 20  CLOSE 1
 30  OPEN "TEST.DAT" AS NEW FILE 1 SIZE 6
 40  FOR J% = 1% TO 100%
 45  S$(I%) = "1234567890" \ I%(J%)=J% \ R(J%)=J%+PI
 50  PRINT #1, S$(J%)
 60  PRINT #1, USING "S###", I%(J%)
 70  PRINT #1, USING "S###.##", R(J%)
 80  NEXT I%
 90  CLOSE 1
100  END
```

	Strings	Integers	Reals	EOF
BYTE SIZE	= 100 [(10 + 2) +	(4 + 2) +	(7 + 2)]	+ 1
	= 100 [27] + 1			
	= 2701			

BLOCK SIZE = 2701/512 = 5.275391 } partial block not allowed; the next higher Integer = 6 Blocks

Note: Looping 94 times instead of 100 permits a SIZE of 5 Blocks.

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.
B0142A-10U8204/SE EN

FLUKE®

Technical Data

1720A Highlighted Learning Program B0143

An Introduction to Virtual Arrays

Virtual Arrays Compared to Main Memory Arrays

The 1720A BASIC provides two types of variable arrays[1] for storing and retrieving data: Main Memory (ordinary or normal) Arrays and Virtual Arrays. These two types of arrays are identical in the following ways:

Identical:

1. Array variables include Reals (floating point), Integers and Strings. Integer array names are followed by %; string array names are followed by $.
2. One or two character array variable names (first character may be any upper case letter; second character may be any upper case letter or any number 0 through 9 or may be omitted). Note that IF, ON, OR, TO, FN, LN, PI and AS are not legal variable names.
3. One and/or two dimensional arrays, i.e., one or two subscripts, enclosed in parentheses (if there are two subscripts they are separated by a comma).
4. The following BASIC statements are corrent for Main Memory Arrays and for Virtual Arrays as well:

Reals	A(1) = B3(5,J%) + Z(8)
Integers	Q2%(0) = DD%(1,1) * E%(I%)
Strings	S$(6) = "A"

   In other words, without additional program statements you cannot know which type of array (Main Memory or Virtual) the above BASIC statements represent.
5. Virtual Arrays can be assigned values from Main Memory Arrays and vice versa; Virtual Arrays can be used in equations with Main Memory Arrays.

1. Refer to Highlighted Learning Program B0142, Main Memory Arrays, for the definition and use of arrays for 1720A BASIC programming; also refer to section 2-27 and section 6 of the BASIC Programming Manual.

Virtual Arrays are different from Main Memory Arrays in the following ways:

Differences:

1. Main Memory Arrays reside in Main Memory. Virtual Arrays temporarily reside in a Main Memory Buffer of 512 bytes (1 block) per channel (file) no.; permanently reside on a storage medium (floppy disk or E-disk).
2. Main Memory Arrays are volatile because Main Memory is volatile. Virtual Arrays survive providing they have been transferred from the Main Memory Buffer to the storage medium (CLOSEing the file insures this).
3. Virtual Arrays are not initialized by the DIM statement. Main Memory Arrays are assigned initial values by the DIM statement.
4. Virtual Arrays are randomly[2] accessed from the storage medium; Main Memory Arrays must be retrieved serially[3] from the file storage medium.
5. Main Memory Arrays require PRINT # < > and INPUT # < > statements to transport data to and from the storage medium. Virtual Arrays are automatically updated on the storage medium as they are used in the program.
6. Main Memory Arrays are created with a DIM or a COM (common main memory for program chaining) statement (COM will not support string variables); Virtual Arrays are created with an OPEN and a DIM # statement.
7. Virtual Arrays can be equivalenced[4] except not with COM statements.
8. Virtual Arrays **do not** require a COM statement in order to be accessed by a chained program. COM is not needed and cannot be used with Virtual Arrays. Main Memory Arrays require a COM statement to survive program chaining.
9. Elements of Virtual Array Strings have a definite, dimensioned length. Elements of Main Memory Array strings are limited in length only by the amount of Main Memory available.

10. Since Main Memory Arrays must share main memory with the BASIC program the maximum amount of Main Memory available for Main Memory Arrays is calculated as follows (assume program occupies 5000 bytes):

    Given: Main Memory = 25,000 Bytes
    − Program Size 5,000 Bytes

    Maximum Array Size = 20,000 Bytes

    On the other hand, each Virtual Array file can be as large as 65,536 Bytes. Total number of Virtual Array files is only limited by available disk storage and the directory limit of 72 files.

11. Main Memory Arrays are stored on the disk as ASCII data; they can be viewed by the File Utility Program. Virtual Arrays are stored as Binary data and cannot be viewed by the File Utility Program.
12. Program execution errors automatically close Virtual Array files, making Virtual Array data inaccessible from the immediate mode, however, this data survives on the storage medium and can be retrieved by a program. Main Memory Arrays are accessible from the immediate mode after a program execution error.
13. Virtual Array data survives a re-RUN or EDIT of the program, but Main Memory Array data is lost in both of these situations.

2. Random Access means each variable's value is accessible without having to count through other variables which exist on the disk ahead of the desired variable.
3. Serial Access means a specific variable is accessible only by counting through the other variables which exist ahead of it on the disk.
4. Equivalenced variables share the same area of main memory.

Defining "Virtual"

After reading the above comparison of Virtual Arrays and Main Memory Arrays, it can be said that Virtual Arrays behave "virtually" the same as if they resided in Main Memory even though they actually reside on the storage medium (floppy disk or E-disk). With the exception of the OPEN, CLOSE and DIM # statements required by Virtual Arrays, the same identical programming code can be used interchangeably for Virtual Arrays or Main Memory Arrays; ignoring for the moment the fact that Virtual Array strings require some additional considerations in some situations due to their fixed length.

Virtual Array Definition

A Virtual Array is a collection of data stored in a random access file storage device, such as the electronic disk or the floppy disk. The data is stored in 1720A internal format (binary) so that no conversion is required during input or output. After a channel has been opened, the Virtual Array is available to the program just like a Main memory Array.

Advantages and Disadvantages

Virtual Arrays can be used to significantly extend the capability of a program. You will probably want to use Virtual Arrays exclusively except in situations where execution speed is critical.

Virtual Array Advantages

1. Non volatile — survives power down of 1720A; survives program chaining and DELETE ALL statements.
2. More Bytes of Main Memory are available for program storage than when Main Memory Arrays are used. Refer to the "TIME" example in the "Programming with Virtual Arrays" lesson for a typical comparison.
3. Random Access means PRINT and INPUT statements **not** necessary for data I/O.
4. Supports Equivalencing.
5. String data can be accessed by chained programs.
6. Text messages can be stored on the storage medium rather than Main Memory. Same text can be used as often as needed.
7. Program can be re-started after a 1720A power down and returned to the exact place in the program where execution ceased (due to powering down).
8. Virtual Arrays can be many times larger than Main Memory Arrays; up to 17 times (over 400K bytes) as much data can exist in Virtual Arrays when a floppy disk and two E-disks are used.

Virtual Array Disadvantages

1. Not allowed in RBYTE or WBYTE statements (see section 7 on IEEE Bus Input and Output statements, in the BASIC Programming Manual).
2. Slower execution than Main Memory Arrays. This difference in execution speeds becomes significant when large amounts of data are being sorted, assigned or operated upon. Exact speed differences are dependent on the application.
3. Unlike Main Memory Arrays where the DIM statement assigns a 0 value to Real and Integer elements and an empty string, i.e., "", to String elements (note CHR$(0) <> ""); newly created Virtual Arrays contain whatever byte arrangement that exists on the storage medium where the arrays reside.

Creating Virtual Arrays

Creating Virtual Arrays requires the following actions:

1. A filename must be associated with the Virtual Arrays.
2. A channel number must be associated with the filename.
3. A determination must be made to use NEW data (create a new disk file for new data) or OLD data (data already in a Virtual Array disk file).
4. The SIZE of the arrays in BLOCKS should be stated (continued upper right of this page).
5. The arrays must be DIMensioned.
6. The DIMension must be associated with the channel number picked in step 2, above.
7. The channel (file) must be CLOSEd in order to transport the most current array data (contained in the BUFFER) to the disk.

Main Memory Array for Comparison

The following example OPENs a file and DIMensions a Main Memory Array, then assigns values to the array and stores it on the disk. Main Memory Array data is stored on the disk in the file named "EXAM1.DAT".

```
10 !"EXAM1.BAS"
100 CLOSE 1                          ! INITIALIZE
110 OPEN "EXAM1.DAT" AS NEW FILE 1 SIZE 1   ! RESERVE AND NAME DISK SPACE
120 DIM A$(5%)
130 FOR I%=0% TO 5%
140    A$(I%)=CHR$(65%+I%)          ! ASSIGN LETTERS "A" THROUGH "F"
150 NEXT I%
160 PRINT #1, A$(0%..5%)            ! STORE ON DISK
170 CLOSE 1                         ! CLOSE FILE
180 END
```

Virtual Array

"EXAM1.BAS" has been altered (to become "EXAM2.BAS") as explained in the program comments, to use a Virtual Array instead of a Main Memory Array. Note line 140 did not change at all and line 160 is no longer needed.

Virtual Array data is stored in Virtual Array File "EXAM2.BIN".

```
10 !"EXAM2.BAS"
100 CLOSE 1                          ! INITIALIZE
110 OPEN "EXAM2.BIN" AS NEW DIM FILE 1 SIZE 1      ! added DIM and .BIN
120 DIM #1, A$(5%)                                 ! added #1,
130 FOR I%=0% TO 5%
140    A$(I%)=CHR$(65%+I%)         ! ASSIGN LETTERS "A" THROUGH "F"
150 NEXT I%
160 !          not needed          PRINT #1, A$(0%..5%)     ! STORE ON DISK
170 CLOSE 1                        ! CLOSE FILE
180 END
```

Retrieving Virtual Arrays

Retrieving Virtual Arrays requires all of the seven steps mentioned in CREATING VIRTUAL ARRAYS except SIZE need not be specified. The operating system already knows the SIZE of the file named. In fact, any attempt to change the SIZE of an OLD file by including a SIZE statement in the OPEN statement, will be ignored by the Operating System. The "NEW" statement is replaced by "OLD" (if "OLD/NEW" is omitted, the program assumes "OLD").

Main Memory Array for Comparison

"EXAM1.BAS" has been altered (to become "EXAM3.BAS") to bring the data from file "EXAM1.DAT") back into Main Memory. Note line 170 is able to access the data from Main Memory even though the file has been closed. Note "NEW" has been changed to "OLD".

```
10 !"EXAM3.BAS"
100 CLOSE 1                         ! INITIALIZE
110 OPEN "EXAM1.DAT" AS OLD FILE 1
120 DIM A$(5%)
130 !
140 INPUT LINE #1, A$(0%..5%) ! INPUT FILE DATA
150 !
160 CLOSE 1                    ! CLOSE FILE
170 PRINT A$(0%..5%)           ! DISPLAY DATA
180 END
```

Virtual Array

"EXAM2.BAS" has been altered (to become "EXAM4.BAS") to retrieve the same Virtual Arrays created by "EXAM2.BAS", using the data stored in Virtual Array File "EXAM2.BIN". Note line 140 treats A$(0..5) **virtually** as if it were in Main Memory.

RUN this program and then, from the immediate mode, type: PRINT A$(0..5) <CR> and note I/O error 313 results because the Virtual Array file has been CLOSEd and A$(0..5) is **not** in Main Memory. Note "NEW" has been changed to "OLD".

```
10 !"EXAM4.BAS"
100 CLOSE 1                       ! INITIALIZE
110 OPEN "EXAM2.BIN" AS OLD DIM FILE 1
120 DIM #1, A$(5%)
130 !
140 PRINT A$(0%..5%)              ! DISPLAY DATA
150 !
160 !
170 CLOSE 1                       ! CLOSE FILE
180 END
```

For more information on Virtual Arrays, refer to 1720A Highlighted Learning Program B0144, "Programming with Virtual Arrays"; also refer to section 6 of the 1720A BASIC Programming Manual.

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0143A-10U8204/SE EN

FLUKE®

Technical Data

1720A Highlighted Learning Program B0144

Programming With Virtual Arrays

Routines No Virtual Array Should Be Without

Although Virtual Arrays behave "virtually" the same as Main Memory Arrays, these arrays do require some special handling to prevent unexpected events such as: loss of data, data not stored on disk, disks that become filled without your knowledge, tabs that don't work, disk garbage appearing as initial array values and strings that won't concatenate because they are filled with "null" characters that you can't see.

The program on the following page illustrates the use of "handlers" which should always be used with Virtual Arrays to prevent these unexpected events.

The purpose of the program will now be explained:

Escape Sequences (lines 100 through 160)

This module provides display enhancements and has no direct effect on the Virtual Array.

Open Disk File to Contain Virtual Arrays
(lines 1000 through 1130)

This module creates the Virtual Array and provides several protections against unexpected results. The "ON CTRL/C" handler (line 1020) guarantees the disk file will always be CLOSEd if the user interrupts the program. This file must be CLOSEd when the program is interrupted in order to:

1. Copy the contents of the buffer onto the disk, insuring that the data won't be lost, and,
2. Delete the previous Virtual Array disk file by having the temporary disk file take its place, insuring no data is lost and that an extra file does not remain on the disk.

Two OPEN statements (lines 1050 and 1070) are used. Line 1050 assumes a data file already exists out on the disk and attempts to OPEN this file. If no file is found, an error is generated and the ON ERROR branch (line 1030) takes over at line 1070 and a NEW file is created, otherwise the OLD file is OPENed. If a NEW file is created, the flag FG% is set (line 1080) to insure the NEW arrays get meaningful initial values (line 1120).

```
10 ! "EXAM5"
20 !
30 ! THIS PROGRAM DEMONSTRATES PROPER PROGRAMMING TECHNIQUES
40 ! FOR A VIRTUAL ARRAY.  THE PROGRAM ALLOWS THE USER TO
50 ! INPUT OR KEEP SIX NAMES OF NO MORE THAN 32 CHARACTERS.
60 ! CHARACTER LENGTHS OF NAMES ARE DISPLAYED.
70 !
100 !***************** ESCAPE SEQUENCES *****************
110 !
120 ES$=CHR$(27%)+"["        ! ESCAPE
130 CH$=ES$+"2J"+ES$+";H"    ! CLEAR SCREEN AND HOME CURSOR
140 EL$=ES$+"K"              ! ERASE TO END OF LINE
150 SL$=ES$+"1K"             ! ERASE TO START OF LINE
160 !
1000 !****** OPEN DISK FILE TO CONTAIN VIRTUAL ARRAYS ******
1010 !
1020 ON CTRL/C GOTO 2150     ! CLOSE FILES BEFORE EXITING PROGRAM
1030 ON ERROR GOTO 1070      ! BRANCH WHEN "NAMES.BIN" DOESN'T EXIST
1035 CLOSE 1                 ! VERIFY CHANNEL 1 HAS NOT BEEN LEFT OPEN
1040 FG%=0                   ! FLAG TO INDICATE NEW DATA
1050 OPEN "NAMES.BIN" AS OLD DIM FILE 1
1060 GOTO 1100               ! JUMP AROUND "NEW" OPEN STATEMENT
1070 OPEN "NAMES.BIN" AS NEW DIM FILE 1 SIZE 1
1080 FG%=1%                  ! INDICATE "NEW" DATA
1090 RESUME 1100             ! EXIT FROM "ON ERROR" BRANCH
1100 OFF ERROR               ! CURRENT ERROR HANDLER NO LONGER NEEDED
1110 DIM #1, NM$(5%)=32%     ! CREATE 6 STRING ELEMENTS OF 32 CHARACTERS EACH
1120 IF FG% THEN GOSUB 20020 \ FG%=0    ! INITIALIZE NEW DATA
1130 !
```

Program Listing is continued on next page

```
2000 !***************** ENTER NAMES *******************
2010 !
2020 ON ERROR GOTO 30020        ! HANDLE INPUT ERRORS
2030 PRINT CH$                  ! CLEAR SCREEN AND HOME CURSOR
2040 FOR I%=0% TO 5%
2050 PRINT CPOS(2,15);"NAME NO.";I%;"IS CURRENTLY ";EL$;NM$(I%)
2060 VA$=NM$(I%)                ! PASS TO MAIN MEMORY VARIABLE FOR SUBROUTINE
2070 GOSUB 21020                ! STRIP OFF NULL CHARACTERS
2080 PRINT CPOS(3,15);"STRING LENGTH =";LEN(VA$),"LEN(NM$(I%)=";LEN(NM$(I%))
2090 PRINT CPOS(7,15);"ENTER NAME <CR>"
2100 PRINT CPOS(5,15);EL$;      ! POSITION CURSOR AND ERASE PREVIOUS ENTRY
2110 INPUT LINE T$
2120 IF T$="" GOTO 2140         ! OPTION TO KEEP PREVIOUS ENTRY
2130 NM$(I%)=T$                 ! TRANSFER ENTRY FROM TEMPORARY STORAGE
2140 NEXT I%
2150 CLOSE 1                    ! COPY BUFFER CONTENTS ONTO DISK
2160 END
2170 !
20000 !***** ASSIGN MEANINGFUL INITIAL VALUES IN LIEU OF DISK GARBAGE *****
20010 !
20020 FOR J%=0% TO 5%
20030   NM$(J%)="YOUR CHOICE"
20040 NEXT J%
20050 RETURN
20060 !
21000 !*************** DELETE NULL CHARACTERS ***************
21010 !
21020 P1%=INSTR(1%,VA$,CHR$(0%))          ! LOCATE 1ST NULL POSITION
21030 SG%=SGN(P1%)                        ! WAS NULL FOUND?
21040 P2%=P1%-1%                          ! LOCATE LAST NON-NULL CHARACTER
21050 VA%=(1%-SG%)*LEN(VA$)+P2%*SG%       ! LENGTH OF CHARACTER STRING W/O NULLS
21060 VA$=LEFT(VA$,VA%)                   ! STRIP OFF NULL CHARACTERS
21070 RETURN
21080 !
30000 !*********** ERROR HANDLER FOR INPUT ERRORS **********
30010 !
30020 PRINT CHR$(7%);CPOS(9,15);"ILLEGAL ENTRY";
30030 WAIT 1000
30040 PRINT SL$;      ! ERASE "MESSAGE"
30050 RESUME 2100
30060 !
32030 END
```

Enter Names (lines 2000 through 2170)

This module provides the display prompts and INPUT LINE statement to allow the user to enter six names. Notice the use of the temporary variable T$ (lines 2110 and 2120). If the user makes no entry before pressing the RETURN key, the BASIC System will input an empty string to T$ and the data contained in NM$ (I%) will remain the same.

The length of NM$(I%) and the relevant string length is displayed (line 2080) to demonstrate the need to strip off the Null Characters (line 2070). Any attempt to concatenate NM$(I%), e.g., NM$(I%) = NM$(I%) + "anything" will result in an error because Virtual Array string elements are always full due to Null Characters which are added by the BASIC System. Proper concatenation could be accomplished as follows:

```
2060 VA$ = NM$(I%)              ! Virtual Array Assigned to
                                  Main Memory Array
2070 GOSUB 21020                ! Strip Off Null Characters⁵
2172 VA$ = VA$ + "anything"     ! Final LEN (VA$) ≤32
2174 NM$ (I%) = VA$             ! Assignment to Virtual Array
```

[5] *This could also be done with a DEF FN (see section 6-22 of the BASIC programming manual). Although somewhat slower than a subroutine, the defined function eliminates the need for dedicated variable names.*

Assign Meaningful Initial Values in Lieu of Disk Garbage
(lines 20000 through 20060)

This subroutine is only used if a NEW data file is OPENed. For an interesting experiment, type: KILL "NAMES.BIN" <CR> from the immediate mode. Then go to FUP and format a new disk and copy your original disk onto the new disk. Using the new disk, get into BASIC and load (OLD) the program. Then delete line 1120. RUN the program and observe the disk garbage. Had Real or Integer arrays been used this module could have assigned zeros to these array elements.

Error Handler for Input Errors

(lines 30000 through 30060)

This subroutine prevents program interruption by handling illegal data entries. RUN the program and make entries which are longer than 32 characters and watch this error handler in action. The branch to this error handler is set-up in line 2020 of the program.

User Information

This program may be interrupted at any time by a CTRL C, and all entries made just prior to the CTRL C will be stored on the disk. In other words, if all entries are correct except the 3rd name, RUN the program and simply press RETURN until the 3rd name is displayed; enter the 3rd name, press RETURN, then CTRL C and you are finished.

Debugging Programs That Use Virtual Arrays

Most programmers will attempt to PRINT variable values from the Immediate Mode after their program has been interrupted by an error. The variable values can give clues as to what went wrong with the program. It is **not** possible to PRINT Virtual Array values from the Immediate Mode after the program has been interrupted by an error, because the BASIC System reacts to the error by CLOSEing all Virtual Array files in order to protect their data. The same problem exists for a CTRL C interrupt when the program contains an ON CTRL/C branch to CLOSE the Virtual Array Files.

Here are two ways to overcome this problem:

1. Use an ON ERROR handler which assigns key Virtual Array values to Main Memory variables and then CLOSEs the Virtual Array files

or

2. Develop the program using Main Memory Arrays. If available Main Memory is lacking, develop a module at a time using Main Memory Arrays. In the previous program all that would have to happen to convert it to Main Memory Arrays is:

   Delete or Comment out (!) lines: 1030 through 1100, 1120 and 2150 Change line 1110 to "DIM NM$(5%)"

After the program has been debugged, add the original versions of the above lines back into the program and take advantage of the Virtual Arrays, once again.

Single Dimension to Double Dimension Equivalencing

The following example shows how to equivalence a single dimension array with a two dimension array having the same number of elements. Delete the "%" from the variable names and you will see it works for real variables as well. It also works for strings.

```
10 ! "EXAM6A" USES INTEGER-INTEGER EQUIVALENCING
20 !          TO EQUIVALENCE ONE AND TWO DIMENSION ARRAYS
30 CLOSE 1
40 OPEN "DATA.BIN" AS NEW DIM FILE 1 SIZE 1
50 DIM #1, A2%(1%,3%)
60 DIM #1, A1%(7%)
70 FOR I%=0% TO 7%
80   READ A1%(I%)    ! READING FOR THE A1%(ARRAY) DOES IT FOR THE A2%(ARRAY)
90 NEXT I%
100 PRINT "DISPLAY ONE-DIM ARRAY"
110 PRINT
120 FOR I%=0% TO 7% \ PRINT A1%(I%);"  "; \ NEXT I%
130 PRINT \ PRINT
140 PRINT "DISPLAY TWO-DIM EQUIVALENCED ARRAY"
150 PRINT
160 FOR J%=0% TO 3%
170   PRINT A2%(0%,J%), A2%(1%,J%)
180 NEXT J%
190 CLOSE 1
200 DATA 100, 200, 300, 400, 500, 600, 700, 800
210 END
```

Integer-String Equivalencing

The following example shows how to create the 16 character string "JOHN FLUKE MFG.?" using 8 integer values. The Virtual String Array A$(0) and the Virtual Integer Array share the same disk space. To create your own message, first assign your message to A$(0), i.e., A$(0) = "Your Message," then PRINT A%(0%..7%) to see what integer values are required.

```
10 ! "EXAM6" USES INTEGER-STRING EQUIVALENCING
20 !          TO SPELL "JOHN FLUKE MFG.?"
30 CLOSE 1
40 OPEN "DATA.BIN" AS NEW DIM FILE 1 SIZE 1
50 DIM #1, A$(0)=16
60 DIM #1, A%(7%)
70 FOR I%=0% TO 7%
80   READ A%(I%)
90 NEXT I%
100 DATA 20298,20040,17952,21836,17739,19744,18246,16174
110 PRINT A$(0)
120 CLOSE 1
130 END
```

Result: "JOHN FLUKE MFG.?" is displayed.

The next example shows how to assign the integer values, 1 through 8 to the Virtual Integer Array A%(0%..7%) using a 16 character Virtual String Array A$(0). These two virtual arrays share the same disk space. To create your own integer numbers, first assign these numbers to A%(0%..7%) then look at each of the 16 characters and determine its ASCII value, e.g.,

FOR I% = 1% TO 16%
PRINT ASCII (MID (A$(0), I%, 1%))
NEXT I%

Place these ASCII values in the DATA statement on line 160.

```
10 ! "EXAM7" USES INTEGER-STRING EQUIVALENCING
20 !          TO ASSIGN A%(0%)=1,A%(1%)=2,...,A%(7%)=8
30 CLOSE 1
40 OPEN "DATA2.BIN" AS NEW DIM FILE 1 SIZE 1
50 DIM #1, A$(0)=16
60 DIM #1, A%(7%)
70 T$=""
80 FOR I%=1% TO 16%
90   READ N%
100  T$=T$+CHR$(N%)
110 NEXT I%
120 A$(0)=T$
130 PRINT "A%(0%..7%) =";A%(0%..7%);
140 PRINT \ PRINT
150 PRINT "A$(0%) = ";A$(0%)
160 DATA 1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0
170 CLOSE 1
180 END
```

Result: "A%(0%..7%) = 1 2 3 4 5 6 7 8" is displayed.
"A$(0%) = ┌⊥┘⺊☒✓ " is displayed.

Creating Dimension Statements Requiring More Than One Program Line

Sometimes one line won't hold all the Virtual Arrays you would like to DIMension. In the examples below the DIM statement started on line 70 is continued on line 80 by equivalencing a dummy array. The dummy array occupies the same memory space as all the arrays on line 70. The dummy array must contain the total number of bytes for all the arrays on line 70 plus the number of bytes, if any, that are left vacant prior to overlapped Block boundaries.

```
10 ! "EXAM8"
20 ! EQUIVALENCING TO ADD TO THE DIMENSION STATEMENT.
30 ! FF%(ARRAY) IS A DUMMY ARRAY EQUIVALENCED TO THE ARRAYS FOR THE
40 ! DIM STATEMENT ON LINE 70
50 CLOSE 1
60 OPEN "DATA3.BIN" AS NEW DIM FILE 1 SIZE 5
70 DIM #1, AA$(10%)=32%,BB$(10%)=32%,CC$(10%)=32%,DD$(10%)=32%,EE$(10%)=32%
80 DIM #1, FF%(5%*11%*32%/2),FF$(10%)=32%
90 CLOSE 1
100 END
```

```
10 ! "EXAM8A"
20 ! EQUIVALENCING TO ADD TO THE DIMENSION STATEMENT.
30 ! HH%(ARRAY) IS A DUMMY ARRAY EQUIVALENCED TO THE ARRAYS FOR THE
40 ! DIM STATEMENT ON LINE 70
50 CLOSE 1
60 OPEN "DATA3.BIN" AS NEW DIM FILE 1 SIZE 3
70 DIM #1, AA$(10%)=32%,BB$(10%)=32%,CC(10%),DD(10%),EE(10%),FF%(10%),GG%(10%)
80 DIM #1, HH%(11%*(32+32+8+8+8+2+2)/2),II%(10%)
90 CLOSE 1
100 END
```

Using the Tab Function With Virtual Array Strings

The Null Characters in Virtual Array Strings cause unexpected events in PRINT and PRINT USING statements which also include the TAB function. In order to avoid this, assign the Virtual Array String element to a Main Memory String variable, strip out the Null Characters from this Main Memory variable then use it in the PRINT or PRINT USING statement in place of the Virtual Array element. Example 9 compares TAB printing with and without Null Characters.

```
10 ! "EXAM9"
20 ! VIRTUAL ARRAYS WITH THE TAB FUNCTION
30 CLOSE 1,2
40 OPEN "KB0:" AS NEW FILE 2
50 OPEN "DATA.VRT" AS NEW DIM FILE 1 SIZE 1
60 !
70 !            COLUMN REFERENCE
80 PRINT "0          1         2         3"
90 PRINT "1234567890123456789012345678 90"
100 !
110 DIM #1, C$(0%)
120 C$(0%)="1234567890"      ! ASSIGN 10 CHARACTERS TO 16 CHARACTER ELEMENT
130 VA$=C$(0%)               ! SUBROUTINE PASS VARIABLE
140 GOSUB 220                ! FIND LENGTH OF STRING W/O NULLS
150 C$=LEFT(C$(0%),VA%)      ! STRIP NULLS
160 PRINT #2,"";TAB(7%);C$;TAB(27%);PI          ! PRINT WITHOUT NULLS
170 PRINT #2
180 PRINT #2,"";TAB(7%);C$(0%);TAB(27%);PI      ! PRINT WITH NULLS
190 CLOSE 1,2
200 END
```

Program Listing is continued on next page

```
210 !************* LENGTH OF STRING W/O NULLS ****************
220 !
230 P1%=INSTR(1%,VA$,CHR$(0%))
240 SG%=SGN(P1%)
250 P2%=P1%-1%
260 VA%=(1%-SG%)*LEN(VA$)+P2%*SG%
270 RETURN
280 END
```

How to Delete <not used> and <temp ent> Files

<not used> and <temp ent> files accumulate on the disk when Virtual Array files are OPENed and not CLOSEd prior to terminating or EDITing a program. These files can only be seen by the File Utility Program (FUP) using the /E command, but they do take up disk memory and can quickly fill a disk. These files can only be deleted by packing the disk (/P command). Note: These files are **not** transferred using the /W command.

Programming For Faster Execution

The following programming techniques can make a significant difference in the time it takes to execute a program.

Open More Channels

A buffer in Main Memory can only contain 1 Block (512 bytes) of a Virtual Array. When your program uses an array element that is stored in a different block than the block in the buffer, the BASIC System must take the block from the buffer and store it on the disk, then copy the other block into the buffer. Every time this happens, the program must wait until the block transfers are complete, resulting in a loss of program execution speed.

Obviously, if we could get more than one block of the Virtual Arrays into Main Memory Buffer, fewer block transfers would be required during program execution, with a resulting increase in execution speed. To accomplish this, split the Virtual Arrays up into as many as 6 separate groups; OPEN a channel number (1 through 6) for each group. This establishes a one block buffer in Main Memory for each channel OPENed. You will, of course, need a separate file name for each channel OPENed. For example:

In place of —

OPEN "DATA.BIN" AS NEW FILE 1 SIZE 11

DIM #1, A (100%), B (100%), C (100%), D (100%), E (100%), F (100%)

use —

OPEN "DATAA" AS NEW FILE 1 SIZE 2
OPEN "DATAB" AS NEW FILE 2 SIZE 2
OPEN "DATAC" AS NEW FILE 3 SIZE 2
OPEN "DATAD" AS NEW FILE 4 SIZE 2
OPEN "DATAE" AS NEW FILE 5 SIZE 2
OPEN "DATAF" AS NEW FILE 6 SIZE 2

DIM#1, A (100%)
DIM#2, B (100%)
DIM#3, C (100%)
DIM#4, D (100%)
DIM#5, E (100%)
DIM#6, F (100%)

Efficient Element Access For Two-Dimensional Arrays

The fastest assignment of values to a large portion of a double subscripted Virtual Array occurs when the program associates the outside loop with the leftmost subscript; and associates the inside loop with the rightmost subscript as shown in the example "TIME" below:

```
4000 ! "TIME"
4010 CLOSE1
4020 OPEN "MF0:TEMP.DAT" AS NEW DIM FILE 1 SIZE 16
4030 DIM #1, A%(63%,63)
4040 T1=TIME
4050 FOR I%=0% TO 63%
4060   FOR J%=0% TO 63%
4070      A%(I%,J%) = 0%
4080   NEXT J%
4090 NEXT I%
```

Program Listing is continued on next page

FLUKE®

```
4100 T2=TIME
4110 PRINT "UNUSED MEMORY = ";MEM
4120 CLOSE 1
4130 PRINT CHR$(7%)
4140 KILL"MF0:TEMP.DAT"
4150 PRINT "TOTAL TIME = ";(T2-T1)/1000;" SEC"
4160 END
```

Refer to the TABLE below for a comparison of execution speeds for both subscript arrangements.

Take Advantage of the Electronic Disk

Storing Virtual Arrays on the E-Disk can make them execute almost as fast as Main Memory Arrays. In the example "TIME" above, change "MF0:" to "ED0:" in lines 4020 and 4140 and note the increase in execution speed.

Note that the Main Memory Arrays in the example "TIME" require 7604 more bytes of Main Memory than the Virtual Arrays.

TABLE OF EXECUTION SPEED AND UNUSED MAIN MEMORY FOR EXAMPLE "TIME"

	Floppy Disk	E-Disk	Main Memory
Program Lines	Use "MF0:" in lines 4020 and 4140	Use "ED0:" in lines 4020 and 4140	Delete 4010, 4020, 4120, 4140; 4030 reads DIM A% (63%,63%)
4070 A%(I%,J%) = 0%	29.12 sec	22.36 sec	21.03 sec
4070 A%(J%,I%) = 0%	461.3 sec	31.53 sec	21.03 sec
Unused Main Memory (MEM)	24274 Bytes	24274 Bytes	16670 Bytes

Use the Same DIM Statement for More Than One Open Statement

If you have groups of data which are identical as in the following example, one DIM # statement is all that is necessary. An error will result if you attempt to execute a second DIM # statement for the same channel no.

```
10 ! "EXAM10"
20 ! MULTIPLE USE OF A SINGLE DIM STATEMENT
30 !
40 CLOSE 1
50 OPEN "DATA4.BIN" AS NEW DIM FILE 1 SIZE 1
60 DIM #1, A$(3%)=64%
70 A$(0%)=CPOS(4,20)+"TEXT CAN BE STORED IN VIRTUAL"
80 A$(1%)=CPOS(5,20)+"ARRAYS FOR USE AS A PROMPT AS"
90 A$(2%)=CPOS(6,20)+"MANY TIMES AS IT IS NEEDED BY"
100 A$(3%)=CPOS(7,20)+"BY YOUR PROGRAM"
110 PRINT A$(0%..3%)
120 CLOSE 1
130 OPEN "DATA5.BIN" AS NEW DIM FILE 1 SIZE 1
150 A$(0%)=CPOS(10,20)+"BY USING THE SAME PROMPT MANY"
160 A$(1%)=CPOS(11,20)+"TIMES, BUT ONLY HAVING TO CODE"
170 A$(2%)=CPOS(12,20)+"IT ONCE AS A VIRTUAL ARRAY, WILL"
180 A$(3%)=CPOS(13,20)+"REQUIRE MUCH LESS MAIN MEMORY"
190 PRINT A$(0%..3%)
200 CLOSE 1
210 END
```

Chaining Programs Which Use Virtual Arrays

Virtual Arrays can be used in lieu of the COM statement, in fact, Virtual Arrays are better because they allow string arrays to survive; COM statements do not. The program "EXAM11", below, is meant to be chained by the program "CHAIN". "EXAM10" was altered as follows to accomplish the chaining:

insert line 205 RUN "EXAM11"

delete lines 110 and 190 in "EXAM10" as they are no longer necessary

```
10 ! "CHAIN"
20 ! MULTIPLE USE OF A SINGLE DIM STATEMENT
30 !
40 CLOSE 1
50 OPEN "DATA4.BIN" AS NEW DIM FILE 1 SIZE 1
60 DIM #1, A$(3%)=64%
70 A$(0%)=CPOS(4,20)+"TEXT CAN BE STORED IN VIRTUAL"
80 A$(1%)=CPOS(5,20)+"ARRAYS FOR USE AS A PROMPT AS"
90 A$(2%)=CPOS(6,20)+"MANY TIMES AS IT IS NEEDED BY"
100 A$(3%)=CPOS(7,20)+"BY YOUR PROGRAM"
120 CLOSE 1
130 OPEN "DATA5.BIN" AS NEW DIM FILE 1 SIZE 1
150 A$(0%)=CPOS(10,20)+"BY USING THE SAME PROMPT MANY"
160 A$(1%)=CPOS(11,20)+"TIMES, BUT ONLY HAVING TO CODE"
170 A$(2%)=CPOS(12,20)+"IT ONCE AS A VIRTUAL ARRAY, WILL"
180 A$(3%)=CPOS(13,20)+"REQUIRE MUCH LESS MAIN MEMORY"
200 CLOSE 1
205 RUN "EXAM11"
210 END
```

```
10 ! "EXAM11"
20 ! CHAINING PROGRAMS WHICH USE EXISTING VIRTUAL ARRAY DATA
30 ! USE "EXAM10" TO CHAIN THIS PROGRAM IN
40 CLOSE 1
50 OPEN "DATA4.BIN" AS OLD DIM FILE 1
60 DIM #1, A$(3%)=64%
110 PRINT A$(0%..3%)
120 CLOSE 1
130 OPEN "DATA5.BIN" AS OLD DIM FILE 1
190 PRINT A$(0%..3%)
200 CLOSE 1
210 END
```

Additional information on program chaining with Virtual Arrays is found in section 10 of the FLUKE BASIC Programming Manual. This section includes an excellent example on how to re-start a specified program after a power interruption using Virtual Arrays.

NOTE: It is not necessary to CLOSE and re-OPEN channels when chaining in a program to the same Virtual Arrays, however, to prevent unexpected results due to errors, it is recommended that the CLOSEing and re-OPENing always be done.

Using Partial Arrays

If you only want the leading portion of the data in a Virtual Array, it is possible to DIMension for just that portion your program requires. "EXAM11" has been altered to only use the first element of the file without disturbing the rest of the arrays in the file. This appears on the next page as "EXAM12".

```
10 ! "EXAM12"
20 ! USING PORTIONS OF VIRTUAL ARRAYS
30 ! USES "EXAM10" VIRTUAL ARRAY DATA
40 CLOSE 1
50 OPEN "DATA4.BIN" AS OLD DIM FILE 1
60 DIM #1, A$(0%)=64%
110 PRINT A$(0%)
120 CLOSE 1
130 OPEN "DATA5.BIN" AS OLD DIM FILE 1
190 PRINT A$(0%)
200 CLOSE 1
210 END
```

Dimensioning for Disk Storage Space Economies

Section 6-24 through 6-33 of the FLUKE BASIC Programming Manual and the Help Lesson "An Introduction to Virtual Arrays" describe why the order that Virtual Array declarations appear in the DIM statement, affects the amount of disk storage required.

The rule of allocating virtual array declarations from left to right in decreasing order of array element lengths is all that is needed to insure disk storage space efficiency, except when a different declaration order is necessary to facilitate tasks such as accessing the leading portion of an array file (see "EXAM12" program, above). The program "EXAM14", below, computes the disk storage space required for all possible orders of DIM statement declaration.

If you need help in visualizing how the bytes per element are allocated for storage on the disk, study the algorythm in "EXAM14" (lines 1160 through 1340). "EXAM14" calculates the most efficient DIMensioning for a group of three Virtual Arrays (see line 750). This program prints out all possible combinations; indicating the disk storage space required for each combination. The program allocates bytes and blocks just as the BASIC System would do it, in the order that the arrays appear in the DIM statement. The printout appears on the 1720A display, however, the printout is easily changed to an RS-232-C port by changing "KB0:" to "KB1:" or "KB2:" in line 450.

"EXAM14" can easily be altered to try combinations for more or less than three arrays. For example, suppose a 4th array "F$(20%,2%)=8%" were added to the existing DIM statement on line 450; these program lines would be updated or added as indicated:

520 N% = 4%
525 NF% = 4% * 3% * 2% * 1%
740 add " ... 4 (order: 1234)"
750 add " ... ,F$ (20%, 2%) = 8%"

Note: Delete comment on line 750 to make room.

add:	822	FOR D% = 1% TO N% ! D% = position of array no. 4
add:	823	IF D% = A% OR D% = B% OR D% = C% GOTO 2165
	825	add \ AP$(D%) = "4"! delete comment to make room
	840	add ... + AP$ (4%)
add:	935	NE%(D%) = 63% \ NY%(D%) = N3% ! Comment
	950	delete "THREE" insert "FOUR"
	960	update for FOUR FACTORIAL (4 x 3 x 2 x 1 = 24 Combinations)
add:	2165	NEXT D%

After RUNning the program for the fourth array, the results for best choice should be: 3240 or 3421.

```
10 ! EXAMPLE 14
20 ! VERSION 1.0, 30 DECEMBER 1981
25 !
26 ! THIS PROGRAM CALCULATES THE SEQUENCE OF VIRTUAL ARRAY DIMENSIONING
27 ! WHICH WILL REQUIRE THE LEAST AMOUNT OF DISK STORAGE
28 !
30 PRINT CHR$(27%)+"[2J"  ! CLEAR DISPLAY
40 !
50 L$="S###.##"          ! PRINT USING FORMAT FOR BLOCKS
60 Y$="S####"            ! PRINT USING FORMAT FOR BYTES
70 !
100 ! DEFINITION OF VARIABLES
110 !
```

Program Listing is continued on next page

```
120 ! (Y) STANDS FOR B(Y)TE IN THE VARIABLE NAMES
130 ! (L) STANDS FOR B(L)OCK IN THE VARIABLE NAMES
140 !
150 ! A% = ORDER SUBSCRIPT
160 ! B% = ORDER SUBSCRIPT
170 ! C% = ORDER SUBSCRIPT
180 ! K% = COUNTER FOR THE SIX POSSIBLE SEQUENCES
190 ! I% = ARRAY NO.
200 ! J% = ELEMENT COUNTER
210 ! L% = BLOCK COUNTER
220 ! N% = TOTAL NO. OF ARRAYS TO BE DIMENSIONED
230 ! Y% = BYTE COUNTER
240 !
250 ! AY% = ACTUAL TOTAL BYTES USED BY ALL THE ARRAYS
260 ! TY% = MINIMUM TOTAL BYTES REQUIRED FOR ALL THE ARRAYS
270 ! AL  = ACTUAL TOTAL BLOCKS USED BY ALL THE ARRAYS
280 ! TL  = MINIMUM TOTAL BLOCKS REQUIRED FOR ALL THE ARRAYS
290 ! MN% = MINIMUM UNUSED BYTES
300 !
310 ! EL%(I%) = ENDING BLOCK NO. FOR ARRAY I%
320 ! EY%(I%) = ENDING BYTE NO. WITHIN BLOCK NO. EL%(I%) FOR ARRAY I%
330 ! NE%(I%) = NO. OF ELEMENTS IN ARRAY I%
340 ! NY%(I%) = NO. OF BYTES PER ELEMENT IN ARRAY I%
350 ! SL%(I%) = STARTING BLOCK NO. FOR ARRAY I%
360 ! SQ$(K%) = SEQUENCE OF DIMENSIONING
370 ! SY%(I%) = STARTING BYTE NO. WITHIN BLOCK NO. SL%(I%) FOR ARRAY I%
380 ! SZ%(I%) = TOTAL NO. OF BYTES REQUIRED BY ARRAY I%
390 ! UB%(K%) = UNUSED BYTES FOR THE KTH SEQUENCE TEST
400 ! UY%(I%) = ACCUMULATOR OF NO. OF UNUSED BYTES IN BLOCKS USED BY ARRAY I%
410 ! AP$(1..N%) = ARRAY POSITION
415 !
420 !**************** PICK OUTPUT DEVICE **************
430 !
440 CLOSE 1
450 OPEN "KBO:" AS NEW FILE 1
460 !
500 !**************** INITIALIZATION ***************
510 !
520 N%=3%       ! NO. OF ARRAYS TO BE DIMENSIONED
525 NF%=3%*2*1% ! = N% FACTORIAL POSSIBLE DIMENSIONING COMBINATIONS
530 RL%=8%      ! NO. OF BYTES REQUIRED BY A REAL NUMBER
540 NT%=2%      ! NO. OF BYTES REQUIRED BY AN INTEGER
550 N1%=2%      ! STRING LENGTH OF 2 CHARACTERS
570 N2%=4%      !                  4
575 N3%=8%      !                  8
580 N4%=16%     !                 16
590 N5%=32%     !                 32
600 N6%=64%     !                 64
610 N7%=128%    !                128
620 N8%=256%    !                256
630 N9%=512%    !                512
640 !
650 DIM SY%(N%),SL%(N%),EY%(N%),EL%(N%),UY%(N%),NE%(N%),NY%(N%),SZ%(N%)
660 DIM SQ$(NF%), UB%(NF%), AP$(N%)
670 K%=0
680 !
700 !********* ASSIGN SUBSCRIPTS FOR THE SIX POSSIBLE SEQUENCES ******
710 !
720 T1$="          SUBSCRIPTS WHICH IDENTIFY ARRAYS"
730 T2$="          --------------------------------"
740 T3$="ARRAY NO.    1         2        3       (ORDER:  123)"
750 T4$=" DIM #1, G%(1%,4%), E(10%,9%), Q$(10%)=64%   ! ARRAYS BEING TESTED"
760 PRINT #1, T1$ \ PRINT #1, T2$ \ PRINT #1, T3$ \ PRINT #1, T4$ \ PRINT #1
770 !
780 FOR A%=1% TO N%             ! A% = POSITION OF ARRAY NO. 1
790 FOR B%=1% TO N%             ! B% = POSITION OF ARRAY NO. 2
800    IF B%=A% GOTO 2180
810 FOR C%=1% TO N%             ! C% = POSITION OF ARRAY NO. 3
820    IF C%=A% OR C%=B% GOTO 2170
825       AP$(A%)="1" \ AP$(B%)="2" \ AP$(C%)="3"   ! ARRAY POSITION STRINGS
830       K%=K%+1%                       ! INCREMENT TEST COUNTER (1% TO NF%)
840       SQ$(K%)=AP$(1%)+AP$(2%)+AP$(3%)   ! STORE ACTUAL SEQUENCE
850 !
```

Program Listing is continued on next page

```
860  ! SUBSCRIPTS IN THE SIX VARIABLES BELOW CORRESPOND
870  ! WITH THE SELECTED DIMENSIONED ORDER BEING TESTED
880  !
890  ! NO. OF ELEMENTS -- BYTES PER ELEMENT -- ORDER --        --  ARRAY NO.
900  !
910      NE%(A%)=10%     \     NY%(A%)=NT%    ! A%TH ARRAY TO BE DIMENSIONED (1)
920      NE%(B%)=110%    \     NY%(B%)=RL%    ! B%TH ARRAY TO BE DIMENSIONED (2)
930      NE%(C%)=11%     \     NY%(C%)=N6%    ! C%TH ARRAY TO BE DIMENSIONED (3)
940  !
950  ! NOTE THERE ARE THREE FACTORIAL POSSIBLE DIMENSION COMBINATIONS
960  ! FOR THE ABOVE THREE ARRAYS: 123, 132, 213, 231, 312, 321
970  !
1000 !*************** MAIN PROGRAM ***********************
1010 !
1020 Y%=0          ! INITIALIZE BYTE COUNTER
1030 L%=1%         ! INITIALIZE BLOCK COUNTER
1040 UB%(K%)=0     ! INITIALIZE TOTAL UNUSED BYTE ACCUMULATOR
1050 !
1060 !                       ----- DISPLAY HEADINGS -----
1070 !
1080 PRINT #1,TAB(34);"ORDER: ";SQ$(K%)
1090 PRINT #1,TAB(9);"ARRAY";TAB(16);"STARTING";TAB(29);"STARTING";
1100 PRINT #1,TAB(44);"ENDING";TAB(57);"ENDING";TAB(70);"UNUSED"
1110 PRINT #1,TAB(9);"NO.";TAB(18);"BYTE";TAB(30);"BLOCK";
1120 PRINT #1,TAB(45);"BYTE";TAB(57);"BLOCK";TAB(70);"BYTES"
1130 !
1140 !                 ---- ALLOCATE STORAGE TO BLOCKS ----
1150 !
1160 FOR I%=1% TO N%               ! ARRAY NO. COUNTER
1170   SZ%(I%)=NE%(I%)*NY%(I%)     ! TOTAL BYTES REQUIRED BY THE ARRAY
1180   SY%(I%)=Y%+1%        ! STARTING BYTE NO.
1190   SL%(I%)=L%           ! STARTING BLOCK NO.
1200   UY%(I%)=0            ! INITIALIZE UNUSED BYTE ACCUMULATOR
1210 !
1220     FOR J%=1% TO SZ%(I%) STEP NY%(I%)   ! ELEMENT COUNTER
1230       IF Y%+NY%(I%)<=512% GOTO 1280     ! BLOCK SIZE EXCEEDED
1240       UY%(I%)=UY%(I%)+512%-Y%           ! ACCUMULATE UNUSED BYTES
1250       Y%=0                              ! INITIALIZE BYTE COUNTER
1260       L%=L%+1%                          ! UPDATE BLOCK NO.
1270       GOTO 1230                         ! START FILLING THE NEW BLOCK
1280       Y%=Y%+NY%(I%)                     ! INCREMENT AN ELEMENT OF BYTES
1290     NEXT J%                             ! GET NEXT ELEMENT
1300 !
1310   EY%(I%)=Y%                ! ENDING BYTE NO.
1320   EL%(I%)=L%                ! ENDING BLOCK NO.
1330   UB%(K%)=UB%(K%)+UY%(I%)   ! ACCUMULATE TOTAL UNUSED BYTES
1340   TY%=TY%+SZ%(I%)           ! ACCUMULATE MINIMUM TOTAL BYTES REQUIRED
1350 !
1360 !                        ---- DISPLAY RESULTS ----
1370 !
1380   PRINT #1,USING Y$,TAB(5);MID(SQ$(K%),I%,1%);TAB(17);SY%(I%);
1390   PRINT #1,USING Y$,TAB(29);SL%(I%);TAB(44);EY%(I%);
1400   PRINT #1,USING Y$,TAB(56);EL%(I%);TAB(69);UY%(I%)
1410 !
1420 NEXT I%               ! GET NEXT ARRAY
1430 !
```

```
2000 !******************* DISPLAY TOTALS ************************
2010 !
2020 AL=EL%(N%)-1%+EY%(N%)/512
2030 TL=TY%/512
2040 AY%=AL*512
2050 PRINT #1, USING Y$,TAB(49);"TOTAL UNUSED BYTES";TAB(69);UB%(K%)
2060 PRINT #1
2070 PRINT #1, USING Y$,TAB(4);N%;" ARRAYS REQUIRING ";TY%;" BYTES IN ";
2080 PRINT #1, USING L$,TL;" BLOCKS"
2090 PRINT #1, USING Y$,TAB(14);"ACTUALLY USE ";AY%;" BYTES IN ";
2100 PRINT #1, USING L$,AL;" BLOCKS"
2110 PRINT #1
```

Program Listing is continued on next page

```
2120 TY%=0
2140 !
2150 !                    ---- GET SUBSCRIPTS FOR NEXT SEQUENCE ----
2160 !
2170 NEXT C%
2180 NEXT B%
2190 NEXT A%
2200 !
3000 !*************** PICK BEST CHOICE *****************
3010 !
3020 MN%=UB%(1%) \ BC$=SQ$(1%)
3030 FOR K%=1% TO NF%
3040   IF UB%(K%)<MN% THEN MN%=UB%(K%) \ BC$=SQ$(K%) \ K1%=K%  ! PICK SMALLEST
3050 NEXT K%
3060 PRINT #1
3070 PRINT #1,TAB(25);"BEST CHOICE IS TO ORDER THE ARRAYS: ";BC$
3080 PRINT #1,TAB(25);"WHICH YIELDS";MN%;"UNUSED BYTES"
3090 PRINT #1
3100 FOR K%=K1%+1% TO NF%
3110   IF UB%(K%)=MN% THEN PRINT "OR ";SQ$(K%)
3120 NEXT K%
3130 CLOSE 1
3140 END
```

Special Consideration for OPEN Statements

It is often convenient to save the names of disk files as string elements in a virtual array. File names used in OPEN statements cannot contain the null characters which all virtual array strings use to fill up unassigned character positions, nor should they contain string functions such as LEFT. Before the contents of a virtual array string element can be used as a file name in an OPEN statement, you must assign its contents to a main memory string and delete the null characters. The main memory string (containing the desired file name without the null characters) should be used in the OPEN statement. Methods to delete null characters are found on page 2 of this lesson.

```
                         OUTPUT FROM "EXAM14"

          SUBSCRIPTS WHICH IDENTIFY ARRAYS
    ARRAY NO.   1          2           3        (ORDER:  123)
     DIM #1, G%(1%,4%), E(10%,9%), Q$(10%)=64%   ! ARRAYS BEING TESTED
                                 ORDER: 123
          ARRAY  STARTING     STARTING      ENDING       ENDING      UNUSED
          NO.      BYTE        BLOCK         BYTE         BLOCK       BYTES
     1              1            1            20            1           0
     2             21            1           392            2           4
     3            393            2           128            4          56
                                                TOTAL UNUSED BYTES     60

         3 ARRAYS REQUIRING  1604 BYTES IN    3.13 BLOCKS
              ACTUALLY USE  1664 BYTES IN    3.25 BLOCKS
                                 ORDER: 132
          ARRAY  STARTING     STARTING      ENDING       ENDING      UNUSED
          NO.      BYTE        BLOCK         BYTE         BLOCK       BYTES
     1              1            1            20            1           0
     3             21            1           256            2          44
     2            257            2           112            4           0
                                                TOTAL UNUSED BYTES     44

         3 ARRAYS REQUIRING  1604 BYTES IN    3.13 BLOCKS
              ACTUALLY USE  1648 BYTES IN    3.22 BLOCKS
                                 ORDER: 213
          ARRAY  STARTING     STARTING      ENDING       ENDING      UNUSED
          NO.      BYTE        BLOCK         BYTE         BLOCK       BYTES
     2              1            1           368            2           0
     1            369            2           388            2           0
     3            389            2           128            4          60
                                                TOTAL UNUSED BYTES     60

         3 ARRAYS REQUIRING  1604 BYTES IN    3.13 BLOCKS
              ACTUALLY USE  1664 BYTES IN    3.25 BLOCKS
                                 ORDER: 312
          ARRAY  STARTING     STARTING      ENDING       ENDING      UNUSED
          NO.      BYTE        BLOCK         BYTE         BLOCK       BYTES
     3              1            1           192            2           0
     1            193            2           212            2           0
     2            213            2            72            4           4
                                                TOTAL UNUSED BYTES      4

         3 ARRAYS REQUIRING  1604 BYTES IN    3.13 BLOCKS
              ACTUALLY USE  1608 BYTES IN    3.14 BLOCKS
                                 ORDER: 231
          ARRAY  STARTING     STARTING      ENDING       ENDING      UNUSED
          NO.      BYTE        BLOCK         BYTE         BLOCK       BYTES
     2              1            1           368            2           0
     3            369            2            64            4          16
     1             65            4            84            4           0
                                                TOTAL UNUSED BYTES     16

         3 ARRAYS REQUIRING  1604 BYTES IN    3.13 BLOCKS
              ACTUALLY USE  1620 BYTES IN    3.16 BLOCKS
                                 ORDER: 321
          ARRAY  STARTING     STARTING      ENDING       ENDING      UNUSED
          NO.      BYTE        BLOCK         BYTE         BLOCK       BYTES
     3              1            1           192            2           0
     2            193            2            48            4           0
     1             49            4            68            4           0
                                                TOTAL UNUSED BYTES      0

         3 ARRAYS REQUIRING  1604 BYTES IN    3.13 BLOCKS
              ACTUALLY USE  1604 BYTES IN    3.13 BLOCKS
```

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

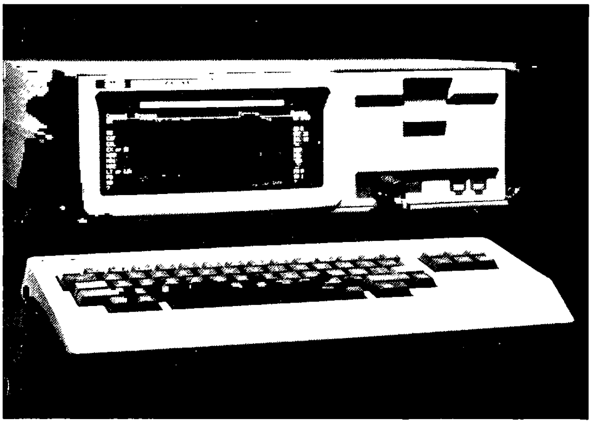
Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.
B0144A-10U8204/SE EN

Technical Data

17XXA Software Information B0090

Stripping Remarks and Extra Spaces From Your Basic Program



Introduction

Many users have faced the dilemma of not having sufficient memory space for running large programs. One reason for this may be that the comments and spaces, required for a well-documented program, use a lot of memory. An additional problem caused by these comments and spaces is an increase in program run time.

One solution has been to run the program in modules. Although this solution works, it is not always the most desirable because it requires additional programming effort. Another solution has been manually to strip the extra spaces and comments from the program listing, thus allowing more program code to reside in main memory. Unfortunately, this solution also consumes and inordinate amount of programmer time and effort.

The purpose of this Software Bulletin is to present a program which will automatically strip remarks and unnecessary spaces from any BASIC program. The resulting program will occupy considerably less memory in most cases, and it will run faster because the comments and extra spaces do not have to be handled by the BASIC interpreter.

The following paragraphs will describe the equipment needed, the program and its limitations, how to enter and test the program, and the benefits of using the program.

Equipment Used

The only equipment needed to run the remark program is the minimum configuration 1720A Instrument Controller, and your program on Floppy or E-disk. Although the E-disk is not required to run the remark program, it will allow the program to run faster.

The Remark Program

The REMARK PROGRAM is written in Fluke 1720A BASIC. The program is divided into several sections. The first sections asks the operator for the input and output file names, then opens the files. The next section reads a line from the input file and strips all of the remarks from it. If a remark is the only thing that occupies a program line, that line will be deleted entirely. The next section of the program strips extra spaces from the remainder of the line and stores the line in the output file. It does not strip spaces from quoted fields, and it strips only invalid spaces from lines containing data statements.

Limitations

If your program contains statements that jump or refer to lines which contain only remarks, the stripped version of your program will produce errors when it encounters those statements. The reason for this is that the referenced lines will have been deleted by the remark program, producing an error when it is run. Therefore, it is a good idea to pre-check your program for all IF-THEN-ELSE, GOTO, GOSUB, STOP ON, ON GOTO, ON GOSUB, TRACE ON, and RESUME statements which refer to remark-only lines. Other program languages allow jumping to labels. Frequently, these labels are the remarks in the referenced line. Although Fluke BASIC allows jumps to these lines, the remark program does not compensate for the fact that such lines will have been deleted.

The remark program will not function properly if the input file which you identify is a lexical, binary, data, virtual array, system, (.SYS) or assembly language (.CIL) file. It will work properly only on BASIC programs which were originally saved on a mass storage device with the "SAVE" command using the BASIC editor. Other types of files will cause fatal or mysterious errors in the remark program.

The object program must have worked properly before running REMARK on it. Otherwise it may contain errors which REMARK assumes do not exist, causing REMARK to modify lines when it otherwise wouldn't. Thoroughly debug your program before running REMARK on it.

The portion of REMARK which deletes extra spaces may inadvertently create errors if the process creates or modifies BASIC keywords. For example, the statement FOR I=S TO P will be changed to FORI=STOP, creating the keyword STOP from the variable S, keyword TO, and variable P. Thus, it is a good idea to run your program through a thorough test cycle after stripping remarks and spaces from it. The Remark Stripping Program is now included in the BASIC startup disk, 1720A-902.

Using the Program

Running the Program

When you run the remark program, the display will ask whether you want to delete only remarks or both remarks and spaces, ask for the input file name (the name of the BASIC program you wish to strip remarks and extra spaces from), and ask the output file name (the name of the stripped version of your program). It will display each line as it is either deleted (as a remark only), or stripped of remarks and extra spaces. When the program is finished, it will display the words "JOB DONE!".

Strip Remarks Only

When the display prompts you to decide whether to strip only remarks or both remarks and spaces (see Figure 1) make one of the following entries:

1. R and press RETURN: strips remarks only;
2. Press RETURN only: strips both remarks and spaces.

```
Remark  Version  1.0
Strips remarks and extra spaces from BASIC programs; does not
correct jumps to remark-only lines deleted by this program !
R=delete remarks only; <RETURN> only= delete remarks and spaces
```

Figure 1. Initial display.

```
When you enter the input <filename.extension>,the .BAS extension
is not needed; invalid filenames or filetypes will cause errors.
Enter input filename =
```

Figure 2. Input file name request.

```
When you enter the output filename, CRT<RETURN> sends results to
the display only. .OUT adds to the input filename if you enter
<RETURN> only, or to output filename if you enter no extension.
Enter output file name =
```

Figure 3. Output file name request.

FLUKE®

b. Test routine after only remarks are stripped.

```
1 DATA
2 DATA \ REM
3 DATA !
4 DATA REM
5 DATA \
100 DATA   45   ,   78   ,   'PRINT' , \REM , "REM" , !!!
102 DATA   DATA   ,   DATA   , 'DATA' , \
104 PRINT \ DATA 4 , " XY " , 7 ' X ' , 8 \ REM ! \ DATA
106 PRINT \ PRINT \ DATA    ,   2 , 23 ,    ,    ,   3
108 DATA   45   ,   78   ,   'PRINT' , \REM , "REM" , !!!
110 PRINT ' HELLO ' , " TODAY ", " 'I' " , ' A " ONE " 2 '
4000 REMOTE
4040 PRINT
4045 PRINT  'ONE',   ' "YES '
```

c. Test routine after both remarks and spaces are stripped.

```
1DATA
2DATA\ REM
3DATA!
4DATAREM
5DATA\
100DATA45   ,78   ,'PRINT',\REM ,"REM",!!!
102DATADATA   ,DATA   ,'DATA',\
104PRINT\DATA4 ," XY ",7 ' X ' ,8 \ REM ! \ DATA
106PRINT\PRINT\DATA,2 ,23 ,,,3
108DATA45   ,78   ,'PRINT',\REM ,"REM",!!!
110PRINT' HELLO '," TODAY "," 'I' ",' A " ONE " 2 '
4000REMOTE
4040PRINT
4045PRINT'ONE',' "YES '
```

Faster Run Time

A stripped version of your program will run faster than an unstripped version. For example, when the unstripped version of the remark program was run to strip the remarks and spaces from that same program, it required nearly 166 seconds of run time. However, when the stripped version of the remark program was run on the unstripped version, it required less than 140 seconds of run time. In both cases the floppy disk was the mass storage device. In this case, stripping the program before running it resulted in a 15% increase in speed.

PROGRAM	Mass Memory Blocks Used	Main Memory Used (Bytes)	Run Time *(Seconds) to strip REMARK.BAS of	
			Remarks	Remarks and Spaces
REMARK.BAS (ASCII, Unstripped)	31	15253	75.83	165.98
REMARK.BAL (lexical, Unstripped)	29	15253	75.83	165.98
REMREM.OUT (ASCII, Stripped of remarks)	14	6073	68.08	146.56
REMSPC.OUT (ASCII, Stripped of remarks and spaces)	8	2994	65.09	139.1
REMSPC.BAL (lexical, Stripped of remarks and spaces)	6	2994	65.09	139.1
*Using floppy as the system device.				

Table 2. STRIPPED and UNSTRIPPED REMARK program memory usage and run times.

For More Information

If you have questions regarding this Software Bulletin or any other Software applications for the 1720A Controller, contact your nearest Fluke sales representative or manufacturing facility.

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0090C-10U8209/SE EN

FLUKE®

Technical Bulletin

1720A Software Information B0091

Converting Binary Data to Floating Point Numbers for 8500A, 8502A, & 8520A

Introduction

The 1720A Instrument Controller can accept readings from the 8500A series of Fluke Programmable Voltmeters in any format. However, only the ASCII (American Standard Code for Information Interchange) form can be used directly in computations with other real numbers. The various binary forms cannot.

The purpose of this Software Bulletin is to provide 1720A software routines which convert the binary formats to real numbers. The routines are provided with remarks to explain them. The actual statements which perform the conversion are minimal, however. One of the main objectives of the routines is to make the conversions as fast as possible. You can incorporate these routines as needed in your present software as explained in this bulletin.

Equipment Needed

The only equipment needed to implement the conversion routines is:

1720A Instrument Controller
Y1720A Programmer's Keyboard

To operate the programs in a system you also need:

8500A, 8502A, or 8520A Voltmeter
with IEEE-488 bus interface
Voltmeter test leads
IEEE-488 bus cable
The device under test

Background

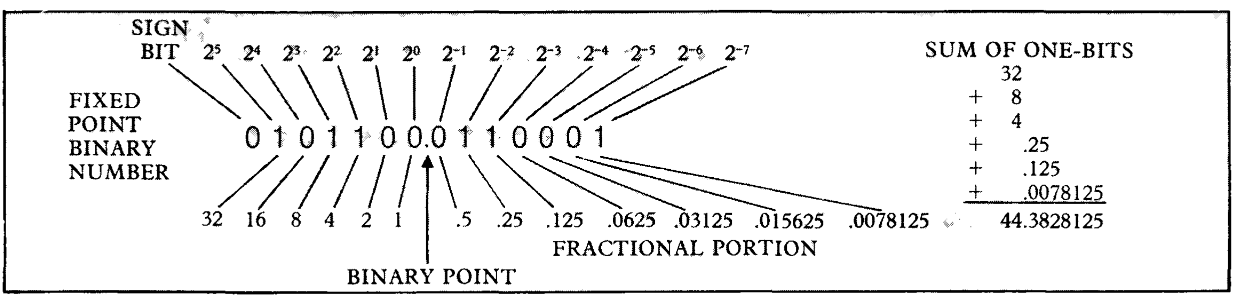
What are the Weights of Binary Digits in Binary Numbers with a Binary Point?

The value of digits to the left of the binary point (integer portion) equals the sum of individual one's multiplied by a positive power of 2. The value of the portion of the number to the right of the binary point (fractional portion) equals the sum of the individual one's multiplied by a negative power of 2. The power of 2 used depends on the bit position. This is shown in Example #1.

What does 2's Complement Mean?

2's complement is a method of representing signed binary numbers. In 2's complement fixed point form, the leftmost bit is a sign bit. If the number is in floating point format, a portion of it will have an exponent and the exponent will have its own sign bit. The value of the whole floating point number is the mantissa times ten-to-the-power-of-the-exponent for the 8500/8502, or two-to-the-power-of-the-exponent for the 8520A.

Example #1



For example, in the below 8502-style binary number system, the exponent of the base 10 and the mantissa is 1.5:

Example #2

Sign Bits

$\underbrace{\underbrace{0010}_{\text{exponent}} \quad \underbrace{01.100}_{\text{mantissa}}}_{\text{Binary}} = \underbrace{1.5 \times 10^2 = 150}_{\text{Decimal}}$

In the below example of the 8520-style binary number system, the exponent of the base 2 and the mantissa is .5:

Example #3

Sign Bits

$\underbrace{\underbrace{0100}_{\text{exponent}} \quad \underbrace{0.100}_{\text{mantissa}}}_{\text{Binary}} = \underbrace{.5 \times 2^4 = .5 \times 16 = 8}_{\text{Decimal}}$

If the sign bit is a zero, the number is a positive value. If the sign bit is a one, the number is a negative value. The difference in the remaining bits is this: **you can determine the value of the base ten positive number by summing up the decimal values of the individual bits (see example #1); but to determine the value of the negative number you must complement the entire number (change all ones to zeros and vise versa) and add 1. Then sum the decimal values of the individual bits (as in example #1) and add a minus sign.** To convert the binary number back to negative form, complement all bits and add 1. Here are examples for both fixed point and floating point numbers.

Example #4

Positive Floating Point Number

	Sign Bits	
Binary:	$\underbrace{0010}$ exponent	$\underbrace{01.011000}$ mantissa
sum decimal values of "one" bits:	2^2	1.375
	1.375×2^2	
Decimal	5.5	

Example #5

Negative Floating Point Number

	Sign Bits	
Binary:	1110 negative exponent (sign bit=1) indicates number is a fraction	10.10100 negative mantissa (sign bit=1)
2's complement: (and add 1)	0001 + 1 0010	01.01011 + 1 01.01100
sum Decimal values of "one" bits:	2^2	1.375
add minus sign	2^{-2}	−1.375
	-1.375×2^{-2}	
Decimal:	−.34375	

Example #6

Positive Fixed Point Number

	Sign Bit
Binary	01001.0010
Sum Decimal values of "one" bits	9.125

Example #7

Negative Fixed Point Number

	Sign Bit
Binary:	10110.111
2's complement (complement and add 1	01001.000 + 1 01001.001
Sum Decimal values of "one" bits:	9.125
add minus sign:	−9.125

FLUKE

What does Sign-Magnitude mean?

Sign-magnitude is another way of representing binary numbers. In this form the leftmost bit is the sign bit. The remaining bits do not have to be complemented or modified in any way; they give the magnitude of the number directly. For example:

$1.011_{(2)} = -.375_{(10)}$, and $0.011_{(2)} = +.375_{(10)}$

A floating point number can have a 2's complement exponent and a sign-magnitude mantissa. Such is the case with 8520A 4-byte (normal mode) readings.

In converting input binary values, the conversion routines follow the philosophy of the above procedure. However, they deviate from the exact procedure in order to provide answers faster with complete accuracy. Further, the routines will only work on other controllers which do integer arithmetic.

General Description

What are the Binary Formats?

The 8500A programmable voltmeters send readings over the IEEE-488 interface in the following binary formats. (Note: There are 8 bits per byte. All readings are in 2's complement form except the 8520A 4-byte binary which has sign-magnitude mantissa and 2's complement exponent.) The formats imply least significant bits to the right.

8520A High Speed Mode
2-Byte Binary
Fixed Point
(multiply by scale factor to get measured value) Reading=3.25

NO EXPONENT

SIGN BIT 00011 010 BYTE 2	ERROR BIT 00000000 BYTE 1

IMPLIED BINARY POINT FOR 10V RANGE

ERROR BIT SETS FOR ± OVERRANGE

8502A High Speed Mode
3-Byte Binary
Fixed Point
(multiply by scale factor to get measured value) Reading=1.75

NO EXPONENT

SIGN BIT ERROR BIT 0001 1100 NOT USED BYTE 1	00000000 BYTE 2	00000000 BYTE 3

IMPLIED BINARY POINT

ERROR OCCURS IF ERROR BIT IS COMPLEMENT OF SIGN BIT
(absolute value of reading ⩾2.0)

8520A Normal Mode
4-Byte Binary
Floating Point
(no scale factor)
(exponent is power of 2)
Reading=.0625

EXPONENT = 2^{-2} MANTISSA = .25

SIGN BIT 11111110 BYTE 1	SIGN BIT 0 0100000 BYTE 2	00000000 BYTE 3	00000000 BYTE 4

IMPLIED BINARY POINT

8500A/8502A Normal Mode
5-Byte Binary
Floating Point
(no scale factor)
(exponent is power of 10)
Reading=35

MANTISSA = 3.5 EXPONENT = 10^{+1}

SIGN BIT 00000011 BYTE 1	10000000 BYTE 2	00000000 BYTE 3	00000000 BYTE 4	SIGN BIT 00000001 BYTE 5

IMPLIED BINARY POINT

NOTE: See topic on "Difference Between Normal and High Speed Reading Modes" for correct Scale factors.

FLUKE®

What does the Controller do with the Binary Values?

When the controller reads the binary values from the voltmeter, it cannot compute with them directly because the controller's internal format for numbers is different from the format received from the voltmeters. For example, the controller assigns 2 bytes of memory for each integer and 8 bytes for each floating point number (also called "real" number), while the voltmeters send 2, 3, 4, or 5 bytes. Since the binary formats of voltmeter readings are so varied and different, the controller must have a program or routine to convert the voltmeter readings into floating point numbers which it can recognize and use in computations.

The routines presented in this software bulletin perform those conversions, and multiply converted values by any necessary scale factors. They also minimize the time required to do the conversions. They then make the converted readings available to the user program in the form of real number (floating point) variables. The user program can use the values of these variables in subsequent computations.

What are the Scale Factors all about?

Readings taken from the voltmeters (8502A and 8520A) in high speed mode are in fixed point format. That is, there is no exponent as part of the reading, and the binary point is assumed always to be in the same position. Therefore, all readings taken in the high speed mode are in the same format, regardless of the range setting of the instrument. Because of this, each reading must be multiplied by a scale factor to adjust the reading for the range setting.

All readings from the 8502A in high speed mode are multiplied by 10 in the conversion routines to normalize them to the 10-Volt Range. Readings from the 8520A do not need to be multiplied by 10 first.

The conversion routines then multiply the 2-byte/3-byte readings by a scale factor which is based on the range setting of the respective voltmeter. The proper scale factors are given in the next topic. The user program must enter a scale factor parameter for the range being used into a local variable before calling the conversion routine.

What is the difference between Normal and High Speed Reading Modes?

In the normal modes, the 8500A, 8502A, and 8520A perform internal calculations to adjust the reading for the selected range. In the high speed mode, to save time, no internal adjustment is made. Therefore, the controller must make the adjustment. Normal mode readings include an exponent; high speed ones do not. Because of the greater number of bytes per reading, normal mode readings have more resolution than high speed ones.

8500A - The 8500A has no high-speed mode available on the IEEE-488 Bus. The reading sent is the actual measured value. It has a 5-byte binary format identical to the 8502A 5-byte format.

8502A - The 8502A has both modes. In normal mode the reading is the actual measured value in 5-byte binary format. In high speed mode the reading is in 3-byte binary format, and is not the actual measured value. To get the actual measured value, the controller must multiply the reading by 10.

The 8502A can take high speed readings on all functions: DC Volts (VDC), AC Volts (VAC), DC Current (IDC), AC Current (IAC), and Ohms. When taking ohms measurements, scale factors must be uniquely computed, then multiplied by the converted measured value. Although the somewhat complex method for computing ohms scale factors is not discussed in this bulletin, complete information is provided by Application Bulletin 25, Appendix A.

Regardless of the type of measurement being taken, (when in binary mode) the reading must be multiplied by a scale factor to yield the actual measured value. The scale factor is computed by the formula SF=10 x RF, where RF (Range Factor) for the different functions is given below. The 3-byte conversion routine automatically performs the times-10 multiplication of the Range Factor once you have put the range factor into local variable RF.

Functions/Range			
VDC	**VAC**	**IAC**	**Scale Factor**
100 mV	---	100 uA	1/64
1V	1V	1 mA	1/8
10V	10V	10 mA	1
100V	100V	100 mA	8
1 kV	1 kV	1A	64

Functions/Range	
IDC	**Scale Factor**
100 uA	−1/64
1 mA	−1/8
10 mA	−1
100 mA	−8
1A	−64

NOTE: Ohms scale factor must be computed from prior ohms measurements. Refer to AB25 for details.

FLUKE

In the high-speed mode, the 7th bit of byte 1 is not used. The 6th bit indicates an error condition (overrange) if different from the sign (8th) bit (absolute value of reading is greater than 2.0).

8520A - The 8520A has both modes. In normal mode the reading is the actual measured value in 4-byte binary format. The reading has a 2's complement exponent and sign-magnitude mantissa. Furthermore, if the exponent equals 128, the entire value equals zero. In high speed mode the reading is in 2-byte binary format, and is the actual measured value only if the 8520A is set to the 10-volt range. On any other range, the reading must be multiplied by below scale factors for DC volts (done by the conversion routine). In this mode, the rightmost bit of byte 1 will equal a 1 if the measured value is beyond the selected range. Scale-factors for ohms measurements are different from those for voltage measurements. Negative scale factors are used to give positive results.

Resistance Range	Scale Factor
10 ohms	-1.5625
100 Ohms	-12.5
1K Ohm	-125
10K Ohms	-1250

DC Volts Range	Scale Factor
100 mV	1/64
1V	1/8
10V	1
100V	8
1000V	64

The Conversion Routines

What are They?

Tables 1 through 4 present listings of subroutines written in Fluke BASIC for the 1720A Instrument Controller. The routines convert voltmeter binary readings which have been input as integer arrays to floating point values recognizable by the 1720A, as follows:

Table 1. 8520A High Speed Mode 2-byte Binary

```
10000 !****************   Subroutine  2BYTE   ***************************
10002 !
10004 !   Version 1.0      20 May 1980   Fluke Control Products Marketing
10006 !
10008 !                    BASIC V1.0, FDOS 1.1
10010 !
10012 !   DESCRIPTION:  2BYTE will convert an integer array < IA% > read
10014 ! in from the 8520A in high speed binary mode (2-byte transfers)
10016 ! into 1720A format and put the answers in result array < RA >.
10018 !   If the error bit is set by the 8520A then that reading is set
10020 ! to zero.
10022 !
10024 !   Format of the 2-byte readings is:
10026 !
10028 ! 1st byte:  bit 0     = +/- overrange error bit
10030 !            bits 1-7 = lower ordered fraction bits (2^-10 TO 2^-4)
10032 ! 2nd byte:  bits 0-2 = higher ordered fraction bits (2^-3 TO 2^-1)
10034 !            Implied Binary Point
10036 !            bits 3-6 = integer bits
10038 !            bit 7    = sign bit
10040 !            bit 8    = EOI bit (end of reading; must be masked out)
10042 !
10044 !   Variables to be initialized before calling:
10046 !
10048 !      IA% = input array with concatenated bytes of all readings
10050 !      IM% = size of input array
10052 !      RA  = result array
10054 !      RF  = range scale factor as follows:
10056 !
10058 !            1=10-V ; 1/64=100-mV ; 1/8=1-V ; 8=100-V ; 64=1-kV
10060 !       -1.5625=10-Ohm ; -12.5=100-Ohm ; -125=1-kOhm ; -1250=10-kOhm
10062 !
```

Table 1. 8520A High Speed Mode 2-byte Binary (Con't.)

```
10064 !   Local Variables:
10066 !
10068 !       B1%,B2%      = bytes 1 and 2 of the reading
10070 !       C1%,C2%      = integer contants for complementing
10072 !       C1,C2        = real constants for the binary conversion
10074 !       I%           = loop counter
10076 !       IS%          = sign of number
10078 !
10100 C1% = 255%                                       ! constant for 1's complement
10110 C2% = 256%                                       ! constant for 2's complement
10120 C1  = 8/RF                                       ! constant for byte 1
10130 C2  = 2048/RF                                    ! constant for byte 2
10140 !
10150 FOR I% = 0% TO IM% STEP 2%                       ! conversion loop
10160 B1% = IA%(I%)                                    ! setup byte 1
10170 IF B1% AND 1% THEN RA(I%/2%)=0 \GOTO 10240       ! error has occurred
10180 B2% = IA%(I%+1%) AND C1%                         ! setup byte 2, strip EOI
10190 IF B2%<128% THEN IS% =1% \ GOTO 10230            ! check for negative reading
10200 IS% = -1%                                        ! set sign to negative
10210 B1% = C2%-B1%                                    ! complement byte 1
10220 B2% = C1%-B2%                                    ! complement byte 2
10230 RA(I%/2%) = (B2%/C1 + B1%/C2) * IS%              ! compute reading
10240 NEXT I%                                          ! loop
10250 !
10260 RETURN
```

Table 2. 8502A High Speed Mode 3-byte Binary

```
11000 !*****************    Subroutine 3BYTE    ******************************
11002 !
11004 !   Version 1.0       20 May 1980     Fluke Control Products Marketing
11006 !
11008 !            System Software: BASIC V1.0, FDOS V1.1
11010 !
11012 !   DESCRIPTION:   3BYTE will convert an integer array <IA%> read in
11014 ! from the 8502A DVM in high speed mode (3-byte binary transfer) to
11016 ! 1720A format, and store the answers in result array <RA>.
11018 !   If the error bit is set by the 8502A then that reading is
11020 ! set to zero.
11022 !
11024 !   Format of the 3-byte reading is:
11026 !
11028 ! 1st byte:  bits 0-3 = upper order fraction bits (2^-4 to 2^-1)
11030 !            Implied Binary Point
11032 !            bit 4    = integer bit
11034 !            bit 5    = error bit (set if absolute reading => 2.0)
11036 !            bit 6    = not used
11038 !            bit 7    = sign bit
11040 ! 2nd byte:  bits 0-7 = middle order fraction bits (2^-12 to 2^-5)
11042 ! 3rd byte:  bits 0-7 = lower order fraction bits (2^-20 TO 2^-13)
11044 !            bit 8    = EOI bit (end of reading; must be masked out)
```

Table 2. 8502A High Speed Mode 3-byte Binary (Con't.)

```
11046 !
11048 !    Variables to be initialized before calling:
11050 !
11052 !       IA% = input array with concatenated bytes of readings
11054 !       IM% = size of input array
11056 !       RA  = result array
11058 !       RF  = range scale factor as follows:
11060 !             1=10-V ; 1/64=100-mV ; 1/8=1-V ; 8=100-V ; 64=1-kV
11062 !
11064 !     Local variables:
11066 !
11068 !       B1%,B2%,B3% = bytes 1, 2, and 3
11070 !       C1%,C2%     = integer constants for complementing
11072 !       C1,C2,C3    = real constants for converting
11074 !       I%          = loop counter
11076 !       IS%         = sign of number
11078 !
11100 C1% = 255%                                 ! constant for 1's complement
11110 C2% = 256%                                 ! constant for 2's complement
11120 C1  = 16/(10*RF)                           ! constant for byte 1
11130 C2  = 4096/(10*RF)                         ! constant for byte 2 (2^12)
11140 C3  = 1048576/(10*RF)                      ! constant for byte 3 (2^20)
11150 !
11160 FOR I% = 0% TO IM% STEP 3%                 ! conversion loop
11170 B1% = IA%(I%)                              ! setup byte 1
11180 B2% = IA%(I%+1%)                           ! setup byte 2
11190 B3% = IA%(I%+2%) AND C1%                   ! setup byte 3; strip EOI
11200 IF B1%<128% THEN IS% =1% \ GOTO 11250      ! check for negative reading
11210 IS% = -1%                                  ! set sign to negative
11220 B1% = C1%-B1%                              ! complement byte 1
11230 B2% = C1%-B2%                              ! complement byte 2
11240 B3% = C2%-B3%                              ! complement byte 3
11250 IF B1% AND 32% THEN RA(I%/3%)=0 \GOTO 11270 ! error has occurred
11260 RA(I%/3%)= (B1%/C1 + B2%/C2 + B3%/C3) *IS%  ! compute result
11270 NEXT I%                                    ! loop
11280 !
11290 RETURN
```

Table 3. 8520A 4-byte Binary

```
12000 !****************    Subroutine 4BYTE    ********************************
12002 !
12004 !    Version 1.0      20 May 1980    Fluke control Products Marketing
12006 !
12008 !         System Software:   BASIC V1.0, FDOS V1.1
12010 !
12012 !    DESCRIPTION:  4BYTE will convert an integer array <IA%> read in
12014 !  from the 8520A in 4-byte mode to 1720A format and store the answers
12016 !  in result array <RA>.
12018 !
12020 !    Format of the 4-byte reading is:
12022 !
12024 !  1st byte:   bits 0-6 = exponent (2^X) bits 2^0 to 2^6
12026 !              bit 7    = exponent sign bit
```

Table 3. 8520A 4-byte Binary (Con't.)

```
12028 ! 2nd byte:  bits 0-6 = upper order fraction bits (2^-7 to 2^-1)
12030 !            Mantissa Implied Binary Point
12032 !            bit 7    = mantissa sign bit
12034 ! 3rd byte:  bits 0-7 = middle order fraction bits (2^-15 to 2^-8)
12036 ! 4th byte:  bits 0-7 = lower order fraction bits (2^-23 to 2^-16)
12038 !            bit 8    = EOI bit (end of reading; must be masked out)
12040 !
12042 !   Variables to be initialized before calling:
12044 !
12046 !      IA% = input array with concatenated bytes of all readings
12048 !      IM% = size of input array
12050 !      RA  = result array
12052 !
12054 !   Local variables:
12056 !
12058 !      B1%,B2%,B4% = bytes 1,2,4 (byte 1 is exponent)
12060 !      C1%,C2%     = constants for complementing/masking
12062 !      C1,C2,C3    = constants for the binary conversion
12064 !      EX          = computed exponent
12066 !      I%          = loop counter
12068 !      IS%         = sign of number
12070 !
12100 C1% = 255%                                  ! 8-bit mask (1's complement)
12110 C2% = 256%                                  ! constant for 2's complement
12120 C2  = 128                                   ! constant for byte 2 (2^7)
12130 C3  = 32768                                 ! constant for byte 3 (2^15)
12140 C4  = 8388608                               ! constant for byte 4 (2^23)
12150 !
12160 FOR I% = 0% TO IM% STEP 4%                  ! conversion loop
12170 B1% = IA%(I%)                               ! setup byte 1
12180 B2% = IA%(I%+1%)                            ! setup byte 2
12190 B4% = IA%(I%+3%) AND C1%                    ! setup byte 4; strip EOI
12200 IF B2%<128% THEN IS% = 1% \ GOTO 12230      ! check for negative reading
12210 IS% = -1%                                   ! set sign to negative
12220 B2% = B2% AND 127%                          ! mask sign bit
12230 IF B1%>127% THEN B1% = (C2%-B1%) *-1%       ! complement exponent
12240 IF B1% THEN EX = 2%^B1%*IS%  ELSE EX=0%     ! compute exponent
12250 RA(I%/4%) = (B2%/C2 +IA%(I%+2%)/C3 +B4%/C4) *EX      ! compute result
12260 NEXT I%                                     ! loop
12270 !
12280 RETURN
```

Table 4. 8500A or 8502A 5-byte Binary

```
13000 !*****************    Subroutine 5BYTE    ******************************
13002 !
13004 !   Version 1.0      20 May 1980     Fluke Control Products Marketing
13006 !
13008 !             System Software:   BASIC V1.0, FDOS V1.1
13010 !
13012 !   DESCRIPTION:  5BYTE will convert an integer array <IA%> read in
13014 ! from the 8500A or 8502A DVM in 5-byte mode to 1720A format, and
```

Table 4. 8500A or 8502A 5-byte Binary (Con't.)

```
13016 ! store the answers in result array <RA>.
13018 !
13020 !   Format for the 5-byte reading is:
13022 !
13024 ! 1st byte:  bits 0-6 = mantissa integer bits 2^0 thru 2^6
13026 !            bit 7    = mantissa sign bit
13028 !            Implied Binary Point
13030 ! 2nd byte:  bits 0-7 = mantissa high order fraction bits (2^-8 to 2^-1)
13032 ! 3rd byte:  bits 0-7 = mantissa medium fraction bits (2^-16 to 2^-9)
13034 ! 4th byte:  bits 0-7 = mantissa lower  fraction bits (2^-24 to 2^-17)
13036 ! 5th byte:  bits 0-7 = exponent (10^X) bits 2^-0 to 2^-6)
13038 !            bit 7    = exponent sign bit
13040 !            bit 8    = EOI bit (end of reading; must be masked out)
13042 !
13044 !   Variables to be initialized before calling:
13046 !
13048 !      IA% = input array with concatenated bytes of all readings
13050 !      IM% = size of input array
13052 !      RA  = result array
13054 !
13056 !   Local variables are:
13058 !
13060 !      B1% - B5%     = bytes 1 through 5 (byte 5 is exponent)
13062 !      C1%,C2%       = constants for complementing & conversion
13064 !      C3,C4         = constants for the binary conversion
13066 !      EX            = computed exponent
13068 !      I%            = loop counter
13070 !      IS%           = sign of number
13072 !
13100 C1% = 255%                                   ! constant for 1's complement
13110 C2% = 256%                                   ! constant for 2's complement
13120 C2  = 256                                    ! constant for byte 2 (2^8)
13130 C3  = 65536                                  ! constant for byte 3 (2^16)
13140 C4  = 16777216                               ! constant for byte 4 (2^24)
13150 !
13160 FOR I% = 0% TO IM% STEP 5%                   ! conversion loop
13170 B1% = IA%(I%)       \   B2% = IA%(I%+1%)     ! setup bytes 1 and 2
13180 B3% = IA%(I%+2%)    \   B4% = IA%(I%+3%)     ! setup bytes 3 and 4
13190 B5% = IA%(I%+4%) AND C1%                     ! setup byte 5; strip EOI
13200 IF B1%<128% THEN IS% =1% \ GOTO 13240        ! check for negative reading
13210 IS% = -1%                                    ! set sign to negative
13220 B1% = C1%-B1%       \   B2% = C1%-B2%        ! complement bytes 1 and 2
13230 B3% = C1%-B3%       \   B4% = C2%-B4%        ! complement bytes 3 and 4
13240 IF B5%>127% THEN B5% = (C2%-B5%) * -1%       ! complement exponent
13250 EX = 10%^B5% * IS%                           ! compute exponent
13260 RA(I%/5%) = (B1%+B2%/C2+B3%/C3+B4%/C4)*EX    ! compute result
13270 NEXT I%                                      ! loop
13280 !
13290 RETURN
```

How do they work with My Program?

Figure 1 shows the necessary structure of your program using the 8520A 2-byte (High Speed Mode) conversion routine. Refer to the figure for the following discussion.

The conversion routines function as callable subroutines. That is, your program accesses one of them with the GOSUB statement. Each routine assumes that your program has read one or more readings of the appropriate number of bytes each from the voltmeter, and concatenated them into the elements of an input array (IA%). Your program must also have dimensioned the input array and the result array for the number of readings taken and selected the correct scale factor.

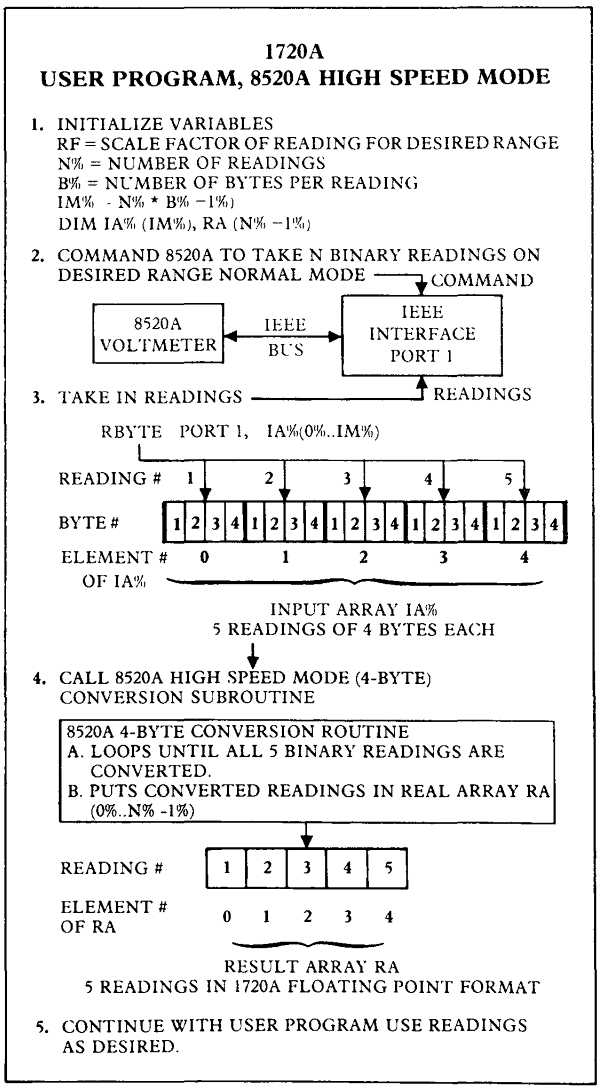


Figure 1. Program Structure to Use Conversion Routines

1. Set variable IM%=N%*B%-1%. This sets up IM% to be the size of integer array IA%; N%= the number of readings to be taken; B%= the number of bytes per reading; 1 is subtracted because array element numbers start at zero.
2. Set variable RF to the scale factor for the range selected This is needed only in 8520A 2-byte and 8502A 3-byte modes.
3. Dimension the readings' input array IA% for the total number of bytes to be read, and the conversion routine result array RA for the number of readings to be taken. The statement DIM IA% (IM%), RA (N%-1%) does this.
4. Program the voltmeter to range and function. Then, read the desired number (N%) of readings into input array IA% (0%..IM%). Readings are concatenated in the array.
5. Call the conversion routine. GOSUB XXXXX does this where XXXXX is the line number of the first executable statement of the subroutine.

The conversion routine takes the bytes from the elements of array IA%, converts them to real number (floating point) format, and puts results in array RA.

If you are using either the 8502A or the 8520A in high speed mode, the conversion routine then multiplies the elements of array RA by the appropriate scale factor to get the correct measurement results. The scale factors are given in the general description. Finally, the conversion routine returns to where your program left off.

The number of readings the conversion routine will handle is determined by the dimensions of the input and result arrays, and thus by available memory. If insufficient memory is available for the number of readings you want to take, your program can set up the result array (RA) as a virtual array. As such, the elements of array RA will be stored in mass memory (Disk or E-Disk). The integer array IA% (input array) cannot be a virtual array because of the way the RBYTE statement is implemented inside the 1720A. The procedure for using virtual arrays is described in the 1720A Programmer's Manual.

How do I enter the Conversion Routine?

1. Select the conversion routine for your application.
2. Ensure that the variables used by the routine selected are not the same as variables intended for other purposes in your program. If they are, change the ones which are easiest to change so that no conflicts occur. If you intend to use more than one conversion routine in your overall program, you may need to rename the variables IA%, IM%, RA and RF. You do not need to change what the listings refer to as LOCAL VARIABLES (unless they interfere with your program) because they are initialized every time the routine is entered.
3. Wherever you want your program to call the subroutine, key in a GOSUB nnnnn statement, where the nnnnn is the starting line number (first executable line which is not a remark) of the subroutine.

Example Programs

Tables 5 through 8 list example programs which use the Binary Conversion Routines in Tables 1 through 4. The programs set up the 8502/8520 DVMs to take 100 voltage readings in the High Speed or Normal mode, put the bytes read into an input array, and display the results. The programs assume that the conversion subroutines are merged into the programs as described in the listing heading. Use the merge command (/M) in the File Utility Program to accomplish this (e.g., DVM2=8520 2, 2 BYTE/M).

Note that the example DVM commands shown between the asterisk lines are examples only. They may not be suitable for your application.

Errors which may occur are overflow error 0 if main memory has too little room for the array of input bytes, and error 306, meaning your mass storage medium has too little room for the result array. To correct the first error reload the program (it should not be merged with any other programs or subroutines than those mentioned in the listing heading). To correct the second error, insert a disk with more space, or purge some of the files.

Other errors may result from incorrect DVM command statements. Consult the DVM manual for correct procedures.

Table 5. 2-byte User Program

```
1  !                         Program 8520 2
2  !
3  !        Version 1.0      20 May 1980      Fluke Control Products Marketing
4  !
5  !           System Software:    FDOS V 1.1, BASIC V 1.0
6  !
10  ! DESCRIPTION:
12  !
14  !     This program will take 100 readings from the 8520A DVM in the 2-byte
16  ! mode (100-volt range), convert them to 1720A format using the 2BYTE sub-
18  ! routine, and display them.  The statements used to command the DVM are
20  ! EXAMPLES only, and are likely not to work in your application.  Key in
22  ! the appropriate statements in their place before running the program.
24  !     The 2BYTE routine is assumed to be stored at line 10000.
26  !
28  !
1000 !*****************     Main Program     ***********************************
1001 !
1010 N% = 100%                                 ! number of readings to take
1020 B% = 2%                                   ! number of bytes per reading
1030 RF = 6                                    ! range factor for 100-V range
1040 IM% = (N% * B%) -1%                       ! size of input array
1050 DIM IA%(IM%), RA(N%-1%)                   ! dimension arrays
1060 !
1070 VM% = 2%                                  ! DVM device number
1080 P% = 0%                                   ! IEEE 488 port number
1090 INIT PORT P%                              ! send IFC and REN to bus
```

Table 5. 2-byte User Program (Con't.)

```
1100 CLEAR PORT P%                                  ! clear bus devices
1110 !
1120 !****************** EXAMPLE STATEMENTS***********************************
1130 ! PRINT @VM%,'VR3I5T0?'                         ! command DVM to take readings   *
1140 ! INPUT @VM%, OB$                               ! clear DVM output buffer        *
1150 ! RBYTE PORT P%, IA%(0%..IM%)                   ! input readings                 *
1160 !                                                                                *
1170 !      NOTE: The above is an example and is given for demonstration             *
1180 ! purposes only.  Because your application may differ, careful                  *
1190 ! considerations is advised before any implementation.  Refer to the            *
1200 ! DVM manual for exact details on how to command the DVM to perform              *
1210 ! its functions.                                                                 *
1220 !**********************************************************************************
1230 !
1240 GOSUB 10100                                    ! call 2BYTE conversion routine
1250 !
1260 FOR I%=1% TO N%                                ! display loop
1270 PRINT I%, RA(I%-1%)                            ! display readings
1280 NEXT I%                                        ! loop till finished
1290 !
1300 END
1310 !
1320 !
10000 !                      Subroutine  2BYTE
```

Table 6. 3-byte User Program

```
1  !                           Program 8502 3
2  !
3  !        Version 1.0      20 May 1980      Fluke Control Products Marketing
4  !
5  !          System Software:   FDOS V 1.1, BASIC V 1.0
6  !
10 ! DESCRIPTION:
12 !
14 !      This program will take 100 readings from the 8502A DVM in the 3-byte
16 ! mode (1-KVolt range), convert them to 1720A format using the 3BYTE sub-
18 ! routine, and display them.  The statements used to command the DVM are
20 ! EXAMPLES only, and are likely not to work in your application.  Key in
22 ! the appropriate statements in their place before running the program.
24 ! Note that the DVM must be externally triggered in the 3-byte mode.
26 !      The 3BYTE routine is assumed to be stored at line 11000.
28 !
30 !
1000 !******************    Main Program    ***********************************
1001 !
1010 N% = 100%                                      ! number of readings to take
1020 B% = 3%                                        ! number of bytes per reading
1030 RF = 64                                        ! range factor for 1KV range
1040 IM% = (N% * B%) -1%                            ! size of input array
1050 DIM BC%(2%), IA%(IM%), RA(N%-1%)               ! dimension arrays
1060 !
```

Table 6. 3-byte User Program (Con't.)

```
1070 P% = 0%                                       ! IEEE 488 port number
1080 VM% = 2%                                      ! DVM device number
1090 BC%(0%) = VM% + 544%                          ! DVM listen address
1100 BC%(1%) = ASCII('?')                          ! DVM trigger command
1110 BC%(2%) = VM% + 576%                          ! DVM talk address
1120 INIT PORT P%                                  ! send IFC and REN to bus
1130 CLEAR PORT P%                                 ! clear bus devices
1140 WAIT 3500                                     ! wait for DVM to clear
1150 !
1160 !****************** EXAMPLE STATEMENTS**********************************
1170 ! PRINT @VM%,'VR4BT2,!'                        ! command DVM to take readings   *
1180 ! INPUT @VM%, OB$                              ! clear DVM output buffer        *
1190 ! RBYTE PORT P%,{WBYTE PORT P%,BC%(0%..2%)} IA%(0%..IM%):3 ! input rdgs *
1200 !                                                                          *
1210 !      NOTE:  The above is an example and is given for demonstration       *
1220 ! purposes only.  Because your application may differ, careful            *
1230 ! considerations is advised before any implementation.  Refer to the      *
1240 ! DVM manual for exact details on how to command the DVM to perform        *
1250 ! its functions.                                                           *
1260 !**************************************************************************
1270 !
1280 GOSUB 11100                                   ! call 3BYTE conversion routine
1290 !
1300 FOR I%=1% TO N%                               ! display loop
1310 PRINT I%, RA(I%-1%)                           ! display readings
1320 NEXT I%                                       ! loop till finished
1330 !
1340 END
1350 !
1360 !
11000 !******************   Subroutine 3BYTE    ********************************
```

Table 7. 4-byte User Program

```
1  !                          Program 8520 4
2  !
3  !         Version 1.0       20 May 1980      Fluke Control Products Marketing
4  !
5  !              System Software:    FDOS V 1.1, BASIC V 1.0
6  !
10 ! DESCRIPTION:
12 !
14 !      This program will take 100 readings from the 8520A DVM in the 4-byte
16 ! mode, convert them to 1720A format using the 4BYTE subroutine, and
18 ! display them.  The statements used to command the DVM are examples only
20 ! and are likely not to work in your application.  Key in the appropriate
22 ! statements in their place before running the program.
24 !      The 4BYTE routine is assumed to be stored at line 12000.
26 !
28 !
```

Table 7. 4-byte User Program (Con't.)

```
1000 !********************    Main Program    ************************************
1001 !
1010 N% = 100%                                ! number of readings to take
1020 B% = 4%                                  ! number of bytes per reading
1030 IM% = (N% * B%) -1%                      ! size of input array
1040 VM% = 2%                                 ! DVM device number
1050 P% = 0%                                  ! IEEE 488 port number
1060 DIM IA%(IM%), RA(N%-1%)                  ! dimension arrays
1070 !
1080 INIT PORT P%                             ! send IFC and REN to bus
1090 CLEAR PORT P%                            ! clear bus devices
1100 !
1110 !***************** EXAMPLE STATEMENTS*********************************
1120 ! PRINT @VM%,'DOI4TO?'                    ! command DVM to take readings   *
1130 ! INPUT @VM%, OB$                         ! clear DVM output buffer        *
1140 ! RBYTE PORT P%, IA%(0%..IM%)             ! input readings                 *
1150 !                                                                          *
1160 !     NOTE: The above is an example and is given for demonstration        *
1170 ! purposes only.  Because your application may differ, careful            *
1180 ! considerations is advised before any implementation.  Refer to the       *
1190 ! DVM manual for exact details on how to command the DVM to perform        *
1200 ! its functions.                                                           *
1210 !**************************************************************************
1220 !
1230 GOSUB 12100                              ! call 4BYTE conversion routine
1240 !
1250 FOR I%=1% TO N%                          ! display loop
1260 PRINT I%, RA(I%-1%)                      ! display readings
1270 NEXT I%                                  ! loop till finished
1280 !
1290 END
1300 !
1310 !
12000 !*****************    Subroutine 4BYTE    ******************************
```

Table 8. 5-byte User Program

```
1 !                         Program 8502 5
2 !
3 !        Version 1.0      20 May 1980     Fluke Control Products Marketing
4 !
5 !            System Software:   FDOS V 1.1, BASIC V 1.0
6 !
10 ! DESCRIPTION:
12 !
14 !     This program will take 100 readings from the 8502A DVM in the 5-byte
16 ! mode (1-KVolt range), convert them to 1720A format using the 5BYTE sub-
18 ! routine, and display them.  The statements used to command the DVM are
20 ! EXAMPLES only, and are likely not to work in your application.  Key in
22 ! the appropriate statements in their place before running the program.
24 !     The 5BYTE routine is assumed to be stored at line 13000.
```

FLUKE®

Table 8. 5-byte User Program (Con't.)

```
26 !
28 !
1000 !*******************   Main Program   *******************************
1001 !
1010 N% = 100%                              ! number of readings to take
1020 B% = 5%                                ! number of bytes per reading
1030 IM% = (N% * B%) -1%                    ! size of input array
1040 DIM IA%(IM%), RA(N%-1%), T%(0%)        ! dimension arrays
1050 !
1060 P% = 0%                                ! IEEE 488 port number
1070 VM% = 2%                               ! DVM device number
1080 T%(0%) = VM% + 576%                    ! DVM talk address
1090 INIT PORT P%                           ! send IFC and REN to bus
1100 CLEAR PORT P%                          ! clear bus devices
1110 WAIT 3500                              ! wait for DVM to clear
1120 !
1130 !***************** EXAMPLE STATEMENTS**********************************
1140 ! PRINT @VM%,'VR4BSOT?'                 ! command DVM to take readings  *
1150 ! WBYTE PORT P%, T%(0%)                 ! command DVM to talk           *
1160 ! RBYTE PORT P%, IA%(0%..IM%)           ! input rdgs                    *
1170 !                                                                        *
1180 !      NOTE:  The above is an example and is given for demonstration     *
1190 ! purposes only.  Because your application may differ, careful          *
1200 ! considerations is advised before any implementation.  Refer to the    *
1210 ! DVM manual for exact details on how to command the DVM to perform     *
1220 ! its functions.                                                         *
1230 !***********************************************************************
1240 !
1250 GOSUB 13100                            ! call 5BYTE conversion routine
1260 !
1270 FOR I%=1% TO N%                        ! display loop
1280 PRINT I%, RA(I%-1%)                    ! display readings
1290 NEXT I%                                ! loop till finished
1300 !
1310 END
1320 !
1330 !
13000 !*****************   Subroutine 5BYTE   ****************************
```

John Fluke Mfg. Co., Inc.
P.O. Box C9090, Everett, WA 98206
800-426-0361 (toll free) in most of U.S.A.
206-356-5400 from AK, HI, WA
206-356-5500 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237

Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0091A-10U8005/SE EN

FLUKE®

Technical Data

Application Information B0101

1720A RS-232-C Interfacing To Serial Printers

RS-232-C Defined

EIA Standard RS-232-C provides the electronics industry with the ground rules necessary for independent manufacturers to design and produce both data terminal and data communication equipment that conforms to a common interface requirement. As a result, a data communications system can be formed by connecting an RS-232-C data terminal to an RS-232-C data communication peripheral (such as a TTY, MODEM, computer, etc.).

The RS-232-C is a hardware standard which guarantees the following:

1. Each device on RS-232-C will use a standard 25-pin connector which will mate to another standard 25-pin of opposite sex.
2. No matter how the cables are connected, no smoke or damage will occur.
3. The data and handshake lines will each be given a specific name.

RS-232-C data and handshake lines:

In serial communications, control and data signals usually come from one pair of lines; an additional line sometimes provides a busy signal - used to delay data transmission until the device can handle that data. The data and handshake lines in RS-232-C send information uni-directionally (simplex); that is, one end of a cable transmits data or handshake and the other end receives data or handshake. Care must be taken to insure that each wire in RS-232-C has the appropriate transmitter and receiver combination. Transmitters connected to transmitters and receivers connected to receivers provide no data communication. To alleviate this problem, the RS-232-C Standard calls out the interface on one end of the cable to be designated as a "Terminal" and the interface on the other end is "Data Communication Equipment". The standard defines the data handshake signals on each pin of the connector for the "Data Communication Equipment" and the "Terminal".

Note: *There is a glossary at the end of this bulletin which can be referred to for most of the terms which may be unfamiliar to the reader.*

RS-232-C Signal Considerations

Timing format conforming to asynchronous operation is shown in figure 1.

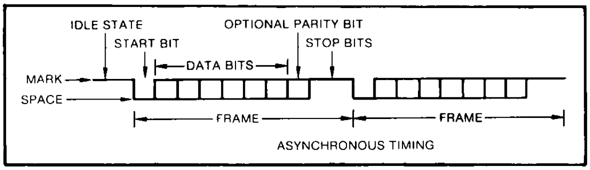


Figure 1. RS-232-C Timing Formats

Asynchronous Operation:

In asynchronous operation each character is bracketed by start and stop bits. These bits separate the characters and synchronize both the transmission and reception of data. When data is not being sent, the transmit line is held high (High=1).

Transmission Mode:

Transmission mode is a fundamental system requirement. It defines the communication ability of both instruments in the system configuration. SIMPLEX indicates data transmission in one direction only. HALF-DUPLEX permits two way communication, but not simultaneously. Simultaneous transmission of data in both directions defines the FULL-DUPLEX system. Obviously, an instrument capable of full-duplex operation can be down graded to simplex operation. However, the reverse is not possible without degrading the system capability.

Baud Rate:

Baud rate is usually selectable on the RS-232-C Interface. If it is not, the manufacturer usually offers a choice when the instrument is purchased. Character format (bits per character and parity) is somewhat flexible between instruments. Investigate the requirement of both instruments before committing either to a system configuration.

Data Interface Levels:

The 1720A uses EIA voltages for data interface levels. EIA voltage levels are: 1 or OFF=−25 to −3V dc, 0 or ON=+3 to +25V dc.

This works fine on paper. However, in practice the user must be aware of the subtleties of serial binary data interchange to ensure that any two pieces of RS-232-C equipment will be compatible.

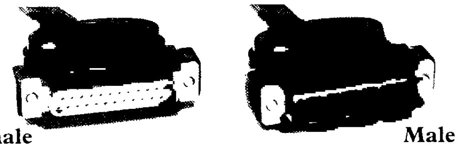
RS-232-C Specification

You can obtain information on this specification from the Electronic Industries Association, Engineering Department, 2001 1st N.W., Washington, D.C. 20006. Send $6.90 per document copy and ask for EIA Standard RS-232: "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange". A companion document, "Industrial Electronics Bulletin No 9 - Application Notes for EIA Standard RS-232-C", costs $3.50.

Printer Considerations for RS-232-C and the 1720A Instrument Controller

The 1720A Instrument Controller has two serial interface ports that meet the requirements of RS-232-C for full duplex asynchronous communication using the EIA voltage levels. The user must verify that the data communication peripheral is pin-for-pin compatible to the 1720A. Table 1 details the RS-232-C cable pinouts for the 1720A and some common printers.

The Y1709 Printer cable has been configured to work directly with the 1776A Printer.The cable has a male connector which mates with the female connector on the 1776A printer. The cable's female connector mates with the male connector on the 1720A. Once the cable has been connected to the 1776A/1720A, the system is ready to be powered up and begin printing. Print statements are given later in this bulletin.



Female　　Male

RS-232-C Connectors

RS-232-C Compatibility Problems

Typical compatability problems associated with RS-232-C are:

1. There are no software standards associated with RS-232-C. Many types of communication protocols serve RS-232-C systems. One protocol uses USASCII code STX (start of text) to precede data and ETX (end of text) to follow data transmission. Another uses USASCII code STX (start of text) to precede data and ETX (end of text) to follow data transmission. Another uses USASCII ACK to acknowledge message receipt and NAK to indicate no acknowledgement. This ACK/NAK combination is usually found in polling computer configurations. (STX, ETX, ACK and NAK are nonprinting characters, for "handshaking" or control only).

Table 1. 1720A TO PRINTER CABLE CONNECTIONS

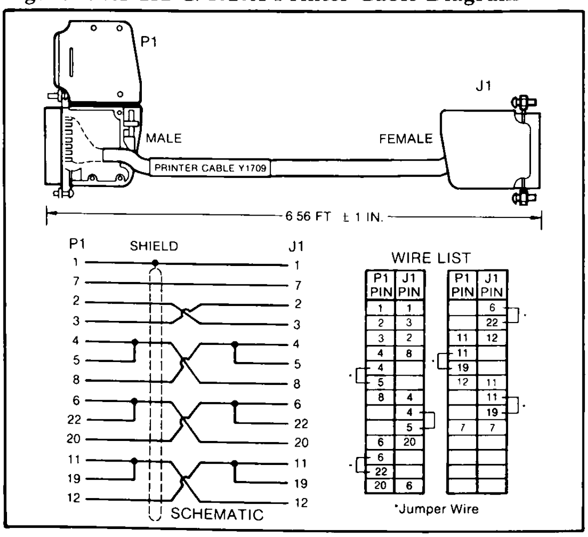
1720A RS-232-C Signal Definitions		Fluke 1776A Tally T1605 T1612	TI 800 Series and Centronics 704	DEC LA120	Lear Siegler 300 Series and Data Royal 5000 7000	IDS 440	Facit 4555 and Anadex DP8000 and Printronix 300	Fluke 2020A	Teletype 43	Printer Signal Definitions
Chassis Ground (Shield)	1	1	1	1	1	1	1	—	1	Chassis Ground (Shield)
Transmitted Data	2 →	3	3	3	3	3	3	2	3	Received Data
Received Data	3 ←	2	2	2	—	—	—	—	—	Transmitted Data
Request to Send	4 →	—	8	—	—	—	—	—	—	Line Signal Detector
Clear to Send	5 ←	—	—	—	—	—	—	—	—	
Data Set Ready	6 ←	20	20	20	20	.	—	—	—	Data Terminal Ready
Signal Common	7 ↔	7	7	7	7	7	7	7	7	Signal Common
Secondary Channel Receive	12 ←	11 or 19	11	11 or 19	19	20	20	6	—	Printer Busy/Ready
Data Terminal Ready	20 →	—	6	—	—	—	—	—	—	Data Set Ready
Cable Connector Type		Male	Male	Female	Male	Female	Male	Male	Female	Female = DB25S Male = DB25P
Comments			TI820 requires a parameter 14- to handle busy							
Fluke Cables		Y1709	Y1709	Y1705 & Y1707	Y1709	N/A	N/A	N/A	Y1705 & Y1707	

2. RS-232-C terminals and RS-232-C data communications equipment are not always hardware compatible.
   A. For example, the two instruments must share at least one of the features from each of the following characteristics:

      1.) Timing Format - Synchronous or asynchronous
      2.) Transmission Mode - Simplex, half-duplex, or full duplex
      3.) Baud Rate (bits per second) - 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 96000, 19200
      4.) Bits per character - 5, 6, 7, or 8
      5.) Parity Bit - Odd, even, high, low, not used
      6.) Data Interface Levels - EIA voltage levels or 20 mA current loop

   B. Care must be taken to ensure that the RS-232-C cable is correct for the application. One of the ambiguous areas in an RS-232-C connection is the use of pin 2 for transmitted data (TD) and pin 3 for received data (RD). The confusion arises in a simplex or half-duplex connection, where pin 2 at one end of the line must go to pin 3 at the other end, and vice versa; this pin transposition can be handled in the cable itself or at either connector. Another confusing aspect of the RS-232-C standard is the pin used to indicate a busy condition. Occasionally pin 11 - normally unassigned - has this task; in other cases pin 19 or pin 20 - with the appropriate polarity - is used.

Figure 2. RS-232-C/1720A Printer Cable Diagram



Consult the manual for a particular device to determine the proper cabling. However, if the manual is not available, the following simple test will tell you if a device is a terminal or data communication device (MODEM) in most cases.

Compatability Test: RS-232-C Cable Application

Measure voltage at pins 2 and 3 with ground lead connected to pin 7.

PERFORM TEST WITH NO CABLES CONNECTED

"TERMINAL"

Pin 2 < −3V Pin 3 0 to +2V Pin 7 GROUND

"DATA COMMUNICATIONS DEVICE" (MODEM)

Pin 2 0 to +2V Pin 3 < −3V Pin 7 GROUND

Obtaining A Printed Program Listing From The 1720A

RS-232-C Printer:

Select an RS-232-C port, connect the printer and set appropriate baud rates

Basic Immediate Mode:

1. Type: OLD "filename" <CR>
2. Type: SAVE "KB1:" <CR>
   or SAVE "KB2:" <CR>

File Utility Program:

Type KB1:=file name <CR>
KB2:=file name <CR>

***Note:** Only ASCII files (.BAS) can be listed in FUP.*

Example:

1720A BASIC program which inputs data from the floppy disk and outputs this data to an RS-232 printer.

```
10 ' PROGRAM      :  LIST.BAS
20 ' PROGRAMMER   :
30 ' REVISION     :  1.0
40 ' DATE         :  15 APR 1980
50 '
60 ' DESCRIPTION :  LISTS AN ASCII FILE ON A RS-232-C PRINTER
70 '
80 '
90                         ON ERROR GOTO 280
100 '
110 '                                          ! ESCAPE SEQUENCE
120 ES$ = CHR$(27%) + "["                      ! RESET SCREEN
130 PRINT ES$;'1p';                            ! PROMPT
140 PRINT ES$;'4;HENTER FILENAME TO BE LISTED' ! STORE FILENAME
150 INPUT A$
160 '                                          ! OPEN CHANNEL FOR INPUT
170 CLOSE 1 \ OPEN A$ AS FILE 1                ! OPEN KB1: FOR OUTPUT
180 CLOSE 2 \ OPEN 'KB1:' AS NEW FILE 2
190 '                                          ! TOP OF FORM
200 PRINT #2,CHR$(12%);
210 !                                          ! INPUT EACH LINE
220 INPUT LINE #1, A$                          ! PRINT EACH LINE
230 PRINT #2, A$                               ! GET NEXT LINE
240 GOTO 220
250 '
260 '                 ERROR HANDLER
270 !                                          ! FILE DOESN'T EXIST
280 IF ERR = 305 THEN RESUME 310               ! END OF FILE
290 IF ERR = 307 THEN RESUME 330
300 PRINT 'ERROR';ERR;'@ LINE';ERL; \ RESUME 330
310 PRINT ES$;'6;HFILE DOESN'T EXIST - TRY AGAIN'; \ GOTO 140
320 '
330 CLOSE 1,2
340 END
```

Glossary

Asynchronous Transmission	Having a variable time interval between successive bits, characters, or events
Baud Rate	The number of bits that can be transmitted in one second
Bit	One of the characters of a two valued or binary number system such as 0 and 1. A bit has come to signify the smallest piece or smallest unit of information. (A single pulse in a group of pulses.)
EIA	Electronic Industries Association 2001 Eye Street, Northwest Washington, D.C. 20006
Format	The predetermined arrangement of characters
Frame	A time period encompassing the bits which define a character
Full Duplex	Simultaneous communication between two points in both directions
Half Duplex	One way communication between two points in either direction
Handshaking	Communication which takes place between two devices for the purpose of informing each other about the status of data being transmitted, received or processed, in order that this may be done in a cooperative, orderly and timely manner without errors. Handshaking is vital to the operation of asynchronous transmission.
Interface, Electrical	Electrical interconnection between system elements
I/O	Input/Output
Modem	A contraction of "modulator-demodulator". In the modem, the square-edged pulse train is impressed (modulated) on a carrier signal of a frequency which is within the telephone channel frequencies between 300 and 3300 Hz. The modem also extracts (de-modulates) the square-edged pulse train from the carrier wave allowing bit-serial communication over standard telephone lines.
Parity	Proper value in the binary check digit of the transmitted and received data.
Parity Bit	A binary digit appended to an array of bits to make the sum of all bits always odd or always even.
Parity Checking	A method used to detect single bit errors
Peripheral Equipment	In a data processing system, any equipment distinct from the central processing unit that may provide the system with outside communication or additional facilities
RS	Recommended standard
Simplex	Data transmission in one direction only
Sync	Short for synchronization
Sync Pulse	An electrical pulse transmitted to a circuit by the master equipment to operate a slave in synchronism with the master.
Synchronous Transmission	A precisely timed bit stream and character stream
TTY	Teletypewriter Equipment

FLUKE®

John Fluke Mfg. Co., Inc.
P.O. Box 43210, Mountlake Terrace, WA 98043
800-426-0361 (toll free) in most of U.S.A.
206-774-2481 from AK, HI, WA and Canada
206-774-2398 from other countries

Fluke (Holland) B.V.
P.O. Box 5053, 5004 EB, Tilburg, The Netherlands
Tel. (013) 673973, TELEX 52237
Phone or write for the name of your local Fluke representative.

Printed in U.S.A.

B0101B-10U8203/SE EN