

Extended BASIC



Extended BASIC

P/N 760496
MARCH 1985

©1985 John Fluke Mfg. Co., Inc.
All rights reserved. Litho in U.S.A.



WARRANTY

John Fluke Mfg. Co., Inc. (Fluke) warrants this instrument to be free from defects in material and workmanship under normal use and service for a period of one (1) year from date of shipment. Software is warranted to operate in accordance with its programmed instructions on appropriate Fluke instruments. It is not warranted to be error free. This warranty extends only to the original purchaser and shall not apply to fuses, computer media, batteries or any instrument which, in Fluke's sole opinion, has been subject to misuse, alteration, abuse or abnormal conditions of operation or handling.

Fluke's obligation under this warranty is limited to repair or replacement of an instrument which is returned to an authorized service center within the warranty period and is determined, upon examination by Fluke, to be defective. If Fluke determines that the defect or malfunction has been caused by misuse, alteration, abuse, or abnormal conditions of operation or handling, Fluke will repair the instrument and bill purchaser for the reasonable cost of repair. If the instrument is not covered by this warranty, Fluke will, if requested by purchaser, submit an estimate of the repair costs before work is started.

To obtain repair service under this warranty purchaser must forward the instrument, (transportation prepaid) and a description of the malfunction to the nearest Fluke Service Center. The instrument shall be repaired at the Service Center or at the factory, at Fluke's option, and returned to purchaser, transportation prepaid. The instrument should be shipped in the original packing carton or a rigid container padded with at least four inches of shock absorbing material. **FLUKE ASSUMES NO RISK FOR IN-TRANSIT DAMAGE.**

THE FOREGOING WARRANTY IS PURCHASER'S SOLE AND EXCLUSIVE REMEDY AND IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE OR USE. FLUKE SHALL NOT BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OR LOSS WHETHER IN CONTRACT, TORT, OR OTHERWISE.

CLAIMS

Immediately upon arrival, purchaser shall check the packing container against the enclosed packing list and shall, within thirty (30) days of arrival, give Fluke notice of shortages or any nonconformity with the terms of the order. If purchaser fails to give notice, the delivery shall be deemed to conform with the terms of the order.

The purchaser assumes all risk of loss or damage to instruments upon delivery by Fluke to the carrier. If an instrument is damaged in-transit, **PURCHASER MUST FILE ALL CLAIMS FOR DAMAGE WITH THE CARRIER** to obtain compensation. Upon request by purchaser, Fluke will submit an estimate of the cost to repair shipment damage.

Fluke will be happy to answer all questions to enhance the use of this instrument. Please address your requests or correspondence to: JOHN FLUKE MFG. CO., INC., P.O. BOX C9090, EVERETT, WA 98206, ATTN: Sales Dept. For European Customers: Fluke (Holland) B.V., P.O. Box 5053, 5004 EB, Tilburg, The Netherlands.

EXTENDED BASIC MANUAL INSTRUCTIONS

Under this instruction sheet you will find the following items:

- ☐ Quick Reference Card.
- ☐ Reference pages beginning with “Array Variable” and ending with “While”.
- ☐ Manual pages for Extended BASIC.

The Quick Reference Card replaces the original one (Form F740) and its update (Form F691). The new card includes all of the commands for IBASIC, CBASIC AND XBASIC. Insert the Quick Reference Card into the BASIC Reference binder.

If your Reference manual has already been updated for XBASIC, discard the Reference pages.

Insert the Reference pages into the BASIC Reference binder according to the number on the upper right hand corner of the reference page. If one of the new reference pages has the same reference number as an existing reference page, the new page replaces the old one. The new pages add keywords that relate to Compiled and Extended BASIC and modify the pages that are unique to Interpreted BASIC.

Insert the Manual pages for Extended BASIC into the empty binder marked “EXTENDED BASIC”.

The Extended BASIC MANUAL is an adjunct to the Interpreted BASIC manual set. When using the Extended BASIC manual, you will need the BASIC manual and the 1722A System Guide available for reference.

Contents

1	GETTING STARTED	1-1
	Introduction	1-3
	Copying A Program Using Tcopy	1-5
	Writing a Program	1-9
	Compiling a Program	1-11
	Linking	1-12
	Running the Program	1-13
	Getting Full Use of Extended BASIC	1-14
	What About Errors?	1-17
	Simplifying the Process	1-18
	Using E-Disk or a Winchester	1-18
	Using Command Files	1-18
	Conclusion	1-18
2	EXTENDED BASIC	2-1
	Introduction	2-3
	The Extended Basic Language	2-4
	Program Development with Extended BASIC	2-6
	Language Names	2-6
	Conventions Used in This Manual	2-7
	Statement Descriptions	2-7
	How to Read Syntax Diagrams	2-8
	Notation Conventions	2-10
	Extended BASIC Syntax	2-11
	Standard Syntax	2-11

Extended Syntax /E Option	2-12
Statement Labels	2-12
Long Variable Names in XBASIC	2-13
Long Variable Names In Extended Syntax	2-14
Continuation Lines	2-15
Omitted Line Numbers /NL Option	2-17
Automatic Integer Conversion /I Option	2-18
Differences in Variables	2-19
Variable Arrays	2-19
ERR\$ System Variable	2-20
Modified Statements	2-21
CALL Statement	2-21
Parameters	2-22
Global Variable and Subroutine Names	2-22
COM Statement	2-24
DIM Statement	2-25
Three Dimensional Arrays	2-25
Array Use in Subroutines	2-25
Conformal Dimensioning	2-26
Redimensioning Main Memory or Common Arrays	2-28
Redimensioning Virtual Arrays	2-28
FOR and NEXT Statements	2-29
ON-GOTO Statements	2-30
REM Statement	2-31
RESTORE Statement	2-31
STOP Statement	2-31
TRACE Statement	2-32
Branch Statements	2-32
Extended BASIC Statements	2-34
SUB Statement	2-35
SUBEND Statement	2-36
SUBRET Statement	2-37
ON-SUBRET Statement	2-38
Control Flow Statements	2-40
Extended IF Statement	2-40
LEAVE Statement	2-42
LOOP Statement	2-43
REPEAT Statement	2-44
SELECT Statement	2-45
WHILE Statement	2-48

Statements Unique To Extended BASIC	2-49
EXPORT Statement	2-49
IMPORT Statement	2-51
Unused Interpreted BASIC Instructions	2-52
 3 EXTENDED BASIC COMPILER	 3-1
Introduction	3-3
Creating Source Code	3-4
Compiling into Object Code	3-4
Linking a Program Together	3-5
Running Extended BASIC Programs	3-6
Memory Allocation in Extended BASIC	3-7
New File Types	3-7
The Extended BASIC Compiler	3-8
Installation Overview	3-8
Installation	3-9
Running the Extended BASIC Compiler Program	3-11
Using the Extended BASIC Compiler Program	3-12
Exiting the Extended BASIC Compiler Program	3-16
Extended BASIC Compiler Options	3-17
Extended Language Syntax: The /E Option	3-17
Integer Conversion: The /I Option	3-17
No Line Numbers: The /NL Option	3-18
No Markers: The /NM Option	3-18
Extended BASIC Compiler Errors	3-19
Linking The Object Files	3-22
Overview of the Linkage Process	3-22
Linking a Program	3-23
Using the Command File	3-25
 4 EXTENDED BASIC RUNTIME SYSTEM	 4-1
Introduction	4-3
Running the Extended BASIC Runtime	
System Program	4-3
Running the Runtime System Program Automatically	4-3
Running the Runtime System Program from	
the Keyboard	4-4
Using the Runtime System Program	4-6
Exiting the Extended BASIC Runtime System	4-7
Runtime System Messages	4-8
Runtime System Error Checking	4-9

5	EXTENDED BASIC LINKAGE UTILITIES	5-1
	Introduction	5-3
	The Extended BASIC Linking Loader Program	5-4
	Executing the XLL Program	5-4
	Terminating the XLL Program	5-4
	Using the XLL Program	5-5
	XLL File Name Conventions	5-7
	Extended Linking Loader Commands	5-8
	END/GO Command	5-10
	FIND Command	5-11
	INCLUDE Command	5-13
	MAP Command	5-15
	OUTPUT Command	5-17
	Extended BASIC Linking Loader Error Messages	5-18
	Extended BASIC Linking Loader Map Format	5-19
	Module List	5-20
	Symbol Table	5-22
	Common Symbol Table	5-24
	Error Message Table	5-24
	The Extended BASIC Library Manager Program	5-25
	Executing the XLM Program	5-25
	Terminating the XLM Program	5-25
	Using the XLM Program	5-26
	XLM File Name Conventions	5-27
	Extended BASIC Library Manager Commands	5-28
	/C - Copy	5-30
	/D - Delete	5-31
	/E - Extended List	5-32
	/L - List	5-34
	/M - Merge	5-35
	/X - Exit	5-36
	? - Help	5-37
	Extended BASIC Library Manager Error Messages	5-38
6	EXTENDED BASIC ERROR MESSAGES	6-1
	Introduction	6-3
	Runtime and Compiler Errors	6-3
	Extended Linking Loader Error Messages	6-8
	Extended Library Manager Error Messages	6-15

APPENDICES

A	Supplementary Syntax Terminology Diagrams	A-1
B	ASCII/IEEE Bus Codes	B-1
C	Extended BASIC Language Software	C-1
D	Reserved Words	D-1
E	Integrating Subroutines from Other Languages	E-1
F	Glossary	F-1

INDEX

Section 1

Getting Started

CONTENTS

Introduction	1-3
Copying A Program Using Tcopy	1-5
Writing a Program	1-9
Compiling a Program	1-11
Linking	1-12
Running the Program	1-13
Getting Full Use of Extended BASIC	1-14
What About Errors?	1-17
Simplifying the Process	1-18
Using E-Disk or a Winchester	1-18
Using Command Files	1-18
Conclusion	1-18

INTRODUCTION

Section 1 of the manual is intended to introduce you to Fluke Extended BASIC (XBASIC) and to give you some hands-on experience. If you have never used a compiled high-level language before, we suggest that you take the time now to read through this material, enter the sample program, and see what the compiler can do for you and your program.

This tutorial is written with the assumption that you are familiar with both the Fluke Instrument Controller and the Fluke BASIC Interpreter program. If you are not, please read the following manuals:

- Getting Started: A New User's Guide to the 1722A Instrument Controller
- BASIC: Sections 1 through 5.

You should also have the 1722A System Guide available for reference.

Compiling a program is only slightly more difficult than writing and executing a program using an interpreter. Executing a compiled program in XBASIC requires four steps:

1. Write the program.
2. Compile the program.
3. Link the program.
4. Run the program.

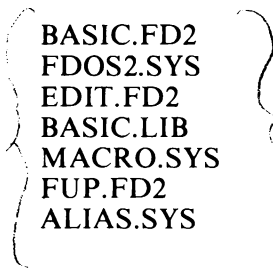
Before you get going, you need to do the following:

1. Make backup copies of:
 - a. The Fluke 1722A System Disk
 - b. The Extended BASIC Disk
2. Put the original copies of the System Disk and the Extended BASIC Disk in a safe place.
3. Copy selected programs from the system disk onto the Extended BASIC disk to create a working disk. All of the files from the system disk will not fit on the XBASIC disk.

To copy the files you need, use either the File Utility Program (FUP) or the Touch-copy program (Tcopy). The File Utility Program requires an extra memory board. It is explained in the 1722A System Guide.

Tcopy is a menu-driven program that utilizes the 1722A's Touch-Sensitive Display to transfer files between the controller's file-structured devices. Tcopy allows you to copy large files quickly, and without a memory board. Tcopy may also be used with only one disk drive. Tcopy is easy to learn and use because the program is totally menu-driven, and it provides on-line Help information. Unlike FUP, Tcopy allows you to select the files to be copied by date, by size, or both. Tcopy is explained in more detail below and in the 1722A System Guide. Tcopy is provided on the System Disk with the file name TCOPY.FD2.

To create your X BASIC working disk, you will need to copy the following files from the System Disk:



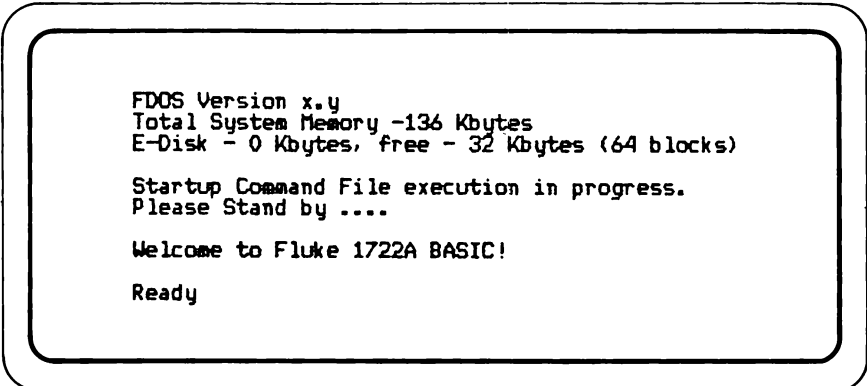
- BASIC.FD2
- FDOS2.SYS
- EDIT.FD2
- BASIC.LIB
- MACRO.SYS
- FUP.FD2
- ALIAS.SYS

Copying a Program Using Tcopy

Tcopy is a simple menu-driven program that can be used to copy whole disks or separate files. This tutorial section will step through the procedures used to create a working disk. We will copy selected files from the System Disk onto the X BASIC Disk, using the INDIVIDUAL option. More information about Tcopy is available on-line by pressing the HELP option after the program is begun.

1. Begin by inserting the backup copy of the System Disk into the disk drive on the Controller and closing the disk drive door. Turn on the 1722A. If it is already on, simultaneously press restart and abort. Remember that pressing these keys also clears the ED0: device.

With the system disk in place, the screen will look like this:



```
FDOS Version x.y
Total System Memory -136 Kbytes
E-Disk - 0 Kbytes, free - 32 Kbytes (64 blocks)

Startup Command File execution in progress.
Please Stand by ....

Welcome to Fluke 1722A BASIC!

Ready
```

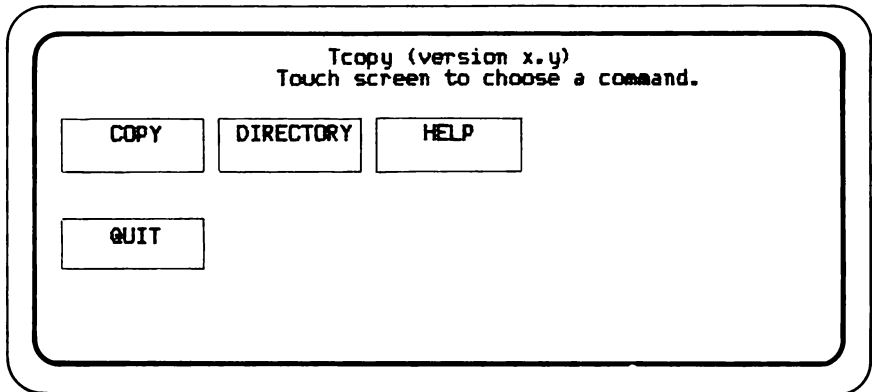
If the version number (x.y) does not match the number on the front of this manual, call a Fluke Customer Service Center for advice.

2. To reach the command level, type

Exit <RETURN>

3. The controller will respond with the FDOS prompt. Type
tcopy <RETURN>

The first menu looks like this:



4. Select the COPY option by touching that box on the screen. The screen will prompt you to select a device to be copied. Select MF and O:, the floppy disk drive, and press CONTINUE. (If you select the wrong command, press the CANCEL box, instead of the START box, and you will return to the main menu.)

5. Tcopy will prompt you to choose a target device. Since you are copying from one floppy disk to another, again select MF and O:.

The screen will now look like this:

Choose a target device.

MF	ED	MB	WD	
0	1	2	3	4
OPTIONS	START	CANCEL		

6. Press OPTIONS. The screen will display the choices: DATE, SIZE, FILES, INDIVIDUAL, START and CANCEL. Select INDIVIDUAL and START.
7. Tcopy will prompt you to insert the source disk and touch the box. The system disk has already been inserted, so just touch the SOURCE box.
8. Tcopy will then ask you to insert the target disk and touch the box. Insert the XBASIC backup disk, close the disk-drive door and touch the TARGET box. Tcopy will prompt you with each file name and request confirmation before copying each file.

Protected files should be copied and all protected files should be overwritten. If a protected file is found on the target disk, Tcopy will ask for permission before overwriting the file.

9. To create a working disk that will allow you to complete the programming sessions in the rest of this section, copy the following files from the System Disk:

BASIC.FD2
FDOS2.SYS
EDIT.FD2
BASIC.LIB
MACRO.SYS
FUP.FD2
ALIAS.SYS

10. Tcopy will prompt you to insert the source disk and touch the SOURCE box. Then you will be prompted to insert the Target disk and touch the TARGET box. Tcopy may repeat this several times, depending on the amount of E-disk memory available.

When Tcopy is finished, you will be returned to the first screen. You can choose to CONTINUE, QUIT, or seek HELP. Working your way through the HELP menus will assist you in learning more about the Tcopy program. Touching QUIT will return you to the FDOS prompt.

CAUTION

If Tcopy is aborted or if the disk is removed from the drive during a copy, some of the files may be lost or the directory of the target disk may be corrupted. Use the FUP/P option to remove any <temp ent> files from the directory. If the directory is unreadable, the disk must be reformatted using the FUP/F option.

Writing a Program

Use the BASIC interpreter's editor to enter and debug this program. Before exiting the BASIC interpreter, run the program and note the time required for execution, then save it as "MF0:SAMPLE.BAS".

Do not remove the disk from the Controller's disk drive until you have finished this exercise.

Enter the BASIC interpreter program by typing

```
FDOS> BASIC
```

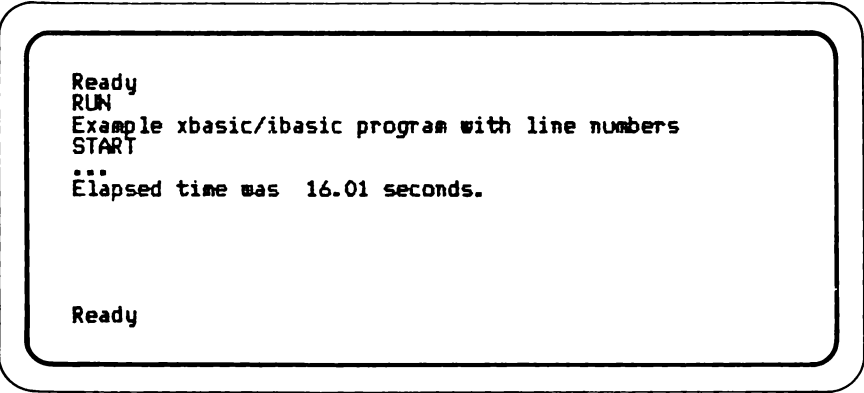
Type in the sample program:

```
5 PRINT "Example xbasic/ibasic program with line numbers"
10 PRINT "START" \ T = TIME
20 K = 0
30 DIM M(5)
40 K = K + 1
45 PRINT K;CHR$(13);
50 A = K/2*3+4-5
60 GOSUB 1000
70 FOR L = 1 TO 5
80 M(L) = A
90 NEXT L
100 IF K < 1000 THEN 40
105 T1 = TIME
110 PRINT "Elapsed time was ";(T1 - T) / 1000; "seconds."
120 END
1000 RETURN
```

To return to the Ready prompt, press <CTRL>/C.

The program computes a number, then loads that number into all elements of a five-element array. This is repeated 1000 times. The loop counter's value is displayed on the screen while program execution time is calculated. The system clock value is stored at the beginning and end of program execution, and some math is performed. Execution time is displayed on the screen at the end of the program.

Before running the program, make sure to clear the Page Mode Button. If the button is lit, the program will not run properly. Type RUN to run the program.



```
Ready
RUN
Example xbasic/ibasic program with line numbers
START
***
Elapsed time was 16.01 seconds.

Ready
```

Using the BASIC interpreter is one way of entering an Extended BASIC program. Other options are possible, especially if you wish to take full advantage of the program statements that are unique to Extended BASIC.

Before exiting the BASIC interpreter to FDOS, save your program by typing

```
Save "MF0: SAMPLE.BAS"
```

Compiling a Program

Now that the sample program has been entered and run, let's compile it. First exit to FDOS and make sure that the mini-floppy disk drive is the System Device by typing the following:

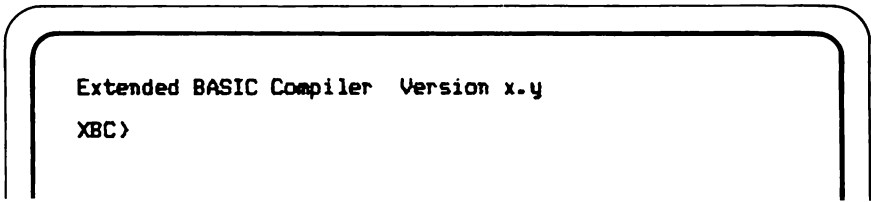
```
FUP MF0:/A
```

Remember to terminate each command line by pressing <RETURN>.

Now run the BASIC Compiler program by typing the following from the FDOS> prompt:

```
XBC
```

Once the BASIC Compiler program is started, it will respond with:

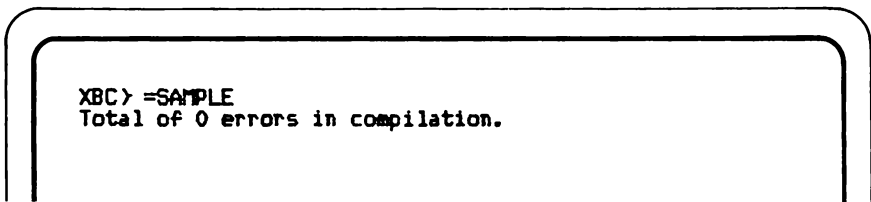
A screenshot of a terminal window with a double-line border. Inside, the text "Extended BASIC Compiler Version x.y" is displayed on the first line, and "XBC>" is on the second line.

```
Extended BASIC Compiler Version x.y  
XBC>
```

Let's compile SAMPLE.BAS. Enter the following line at the XBC> prompt:

```
=SAMPLE
```

The compiler should respond like this:

A screenshot of a terminal window with a double-line border. Inside, the text "XBC> =SAMPLE" is on the first line, and "Total of 0 errors in compilation." is on the second line.

```
XBC> =SAMPLE  
Total of 0 errors in compilation.
```

What you just did was to compile the BASIC program SAMPLE.BAS, putting the object code (compiler output) into the file "MF0:SAMPLE.OBX".

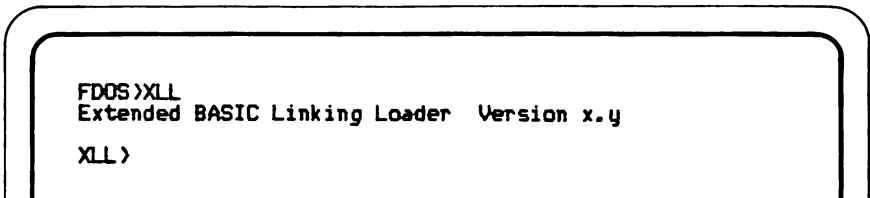
Since the compiler program returned zero errors in compilation, you can assume that your program compiled correctly. The next step is linking.

Linking

We will link the program SAMPLE.BAS with the Linking Loader program, XLL.FD2. At this point, you should have the FDOS prompt on the screen. Run the Linking Loader program by typing:

XLL

After the XLL program is loaded into the Controller's memory, the screen should look like this:



```
FDOS>XLL
Extended BASIC Linking Loader  Version x.y
XLL>
```

In response to the XLL prompt, enter each of the following command lines:

```
INCLUDE SAMPLE
OUTPUT SAMPLE
MAP
GO
```

The XLL program now begins the loading process. The first command line tells XLL to use the object file created by XBC as its input. The second line tells XLL to name the output file SAMPLE.FD2. The third line tells XLL to display the load map. The fourth line tells XLL to begin. When XLL finishes, control will return to FDOS.

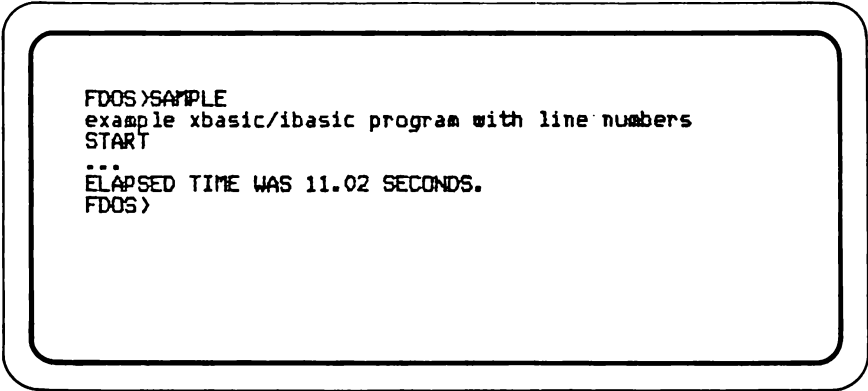
The XLL program has now converted your program into machine executable format, added the header information that allows the Extended BASIC Runtime System program to be loaded automatically, and written the completed file on the disk. The load map, with program information, is displayed and the program is ready to run. See Section 5 for a complete discussion of the load map.

Running the Program

Running the program is simple. From the FDOS> prompt, type:

SAMPLE

The screen should look like this:



```
FDOS>SAMPLE
example xbasic/ibasic program with line numbers
START
...
ELAPSED TIME WAS 11.02 SECONDS.
FDOS>
```

Compare the execution time for the compiled program with the execution time when the program was run by the BASIC interpreter. Which is faster?

GETTING FULL USE OF EXTENDED BASIC

By now you have seen how easy it is to compile a BASIC program written for use with the BASIC interpreter. The example program showed a clear-cut speed advantage after compilation and linking. These are only some of the advantages offered by the Extended BASIC Language.

Taking full advantage of the features offered by Extended BASIC requires new statements and a slightly different syntax. The new syntax is called Extended Syntax. It permits the BASIC programmer to use variable names that are longer than two characters and permits program labels instead of line numbers for branch instruction targets (like GOTO and GOSUB). Other syntactic options are offered as well, and are discussed elsewhere in this manual.

Let's look at a few of the advantages offered by the extended syntax option and Extended BASIC's control flow statements. The Getting Started Disk contains an IBASIC program stored as the source file, LACE.BAS. You may use this program to compare the readability of Extended BASIC'S extended syntax.

Create a disk file containing the following program using the System Editor program (EDIT.FD2). If you aren't familiar with the System Editor, you can find a good tutorial in Section 6 of the 1722A System Guide.

- If you don't like typing, omit the lines beginning with "!".
- The keywords and statements must be entered exactly as shown. The indentations are for visual clarity and need not be entered as shown.

At the FDOS> prompt, type

```

EDIT LACE1.BAS.

! clear screen and graphics plane
PRINT CHR$(27); "[2J"
ERACRP(0%) \ GRPDM

! initialize constants
xcenter% = 320% \ ycenter% = 112%
xscale=120 \ yscale = 100
deg_to_rad = PI / 180

LOOP

! make 20 different variations
FOR no_of_points% = 2% TO 21%
  ERACRP(0%)
  ! calculate number of degrees between points - angle_inc
  angle_inc = 360 / no_of_points%

  ! for each point
  FOR curr_point% = 0% TO no_of_points% - 1%
    ! calculate radian angle of current point - angle
    angle = curr_point% * angle_inc * deg_to_rad
    ! compute x and y coordinates of current point
    curr_x% = xcenter% + xscale * SIN(angle)
    curr_y% = ycenter% + yscale * COS(angle)

    ! from next point to the rest of the points
    FOR next_point% = curr_point% TO curr_point% + no_of_points%
      ! move to current point
      MOVE(curr_x%, curr_y%)
      ! calculate angle from current point to next point
      next_angle = next_point% * angle_inc * deg_to_rad
      ! calculate next x and y
      nextx% = xcenter% + xscale * SIN(next_angle)
      nexty% = ycenter% + yscale * COS(next_angle)
      ! draw line to next point
      PLOT(nextx%, nexty%, 1%)
    NEXT next_point%

  NEXT curr_point%

NEXT no_of_points%

! start all over again
ENDLOOP

```

Now that you've entered the program, notice how readable the long variable names make it. Next, notice how the indented block structure makes it easy to associate each of the nested FOR statements with its own NEXT statement. Last, look at the use of the LOOP-ENDLOOP statements to create a loop with 20 variations.

Save the file by typing

```
<ESC> :W <RETURN>
```

Exit the editor by typing

```
<ESC> :Q <RETURN>
```

Compile LACE1.BAS by typing

```
FDOS> XBC  
XBC> =LACE1
```

Oops! Look at all those error messages! What's wrong? Ooh-ooh I forgot the command-line options that tell the compiler that the source program used extended syntax and no line numbers. Try again.

```
XBC  
XBC> =LACE1/E/NL
```

Yes, that was it. No compiler errors this time. Now to link it. Use the linking loader to link and load the object file.

```
XLL  
XLL> I LACE1  
XLL> O LACE1  
XLL> FIND BASIC  
XLL> G
```

If everything went right, you should have the FDOS prompt on the display again. If so, run the new program:

```
FDOS> LACE1
```

Type <CTRL>/C to exit LACE1.

NOTE

It's impossible to even attempt to show the expected program results in printed form.

What About Errors?

Typically, when the compiler detects an error, an error message appears on the display indicating the line number where the error occurred. If an error does occur, take note of the line numbers displayed. Then use the System Editor's "G" command to go to that line and correct the error. An error that occurs during the linkage process will either appear in the linkage utility program's error file or on the display.

It's not likely that an error will appear in the sample program because it was written and debugged using the BASIC interpreter. If the program were written using the system editor, the BASIC interpreter would not check the syntax. Syntax errors would be found by the compiler. As an experiment, try introducing a deliberate error, such as commenting out the NEXT statement or dropping a quote mark from a PRINT statement.

The compiler's error messages are listed in Section 6 and on the Quick Reference Card. Some of these are considerably different from those issued by the BASIC interpreter. The quantity of error messages issued by the compiler can also vary greatly, depending on the nature of the error. For example, forgetting to include the /NL command-line option will cause a flurry of error messages, even from a small program.

Simplifying the Process

While the actual processes involved are not tremendously complicated, there is a certain amount of time spent typing command lines into the Controller or just waiting for programs to finish running. Here are some suggestions that can minimize some of the delays and relieve some of the tedium of repeatedly typing the same thing over and over.

Using E-Disk or a Winchester

Long programs may take significant amounts of time to compile and link. Much of this time is spent writing the various output files onto the mass-storage device. You can save time by using a storage medium with high I/O throughput capability, such as the E-Disk or the optional Winchester fixed disk drive.

Using Command Files

Typing information into each program at the appropriate time consumes a lot of time when compiling and linking a program. A command file, written for a specific program, takes the drudgery out of typing the same thing, over and over again as a program is developed and debugged. Use the system editor to create the command file. A working command file is supplied on the Extended BASIC distribution disk. This command file (XBCC.CMD) can be used to compile a BASIC program originally written for Interpreted BASIC. Other command files may be found on the 17XXA Instrument Controller System disk, in the System Guide, and at the end of Section 3 of this manual.

Conclusion

You have learned how to use Tcopy to copy individual programs, and have presented and explained the command lines for the compiler and linker programs. We have compared the increased speed in execution of a compiled and linked program to the same program run by an interpreter. Finally, we have explored a few of the features that are unique to the BASIC compiler.

Section 2

Extended BASIC

CONTENTS

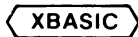
Introduction	2-3
The Extended Basic Language	2-4
Program Development with Extended BASIC	2-6
Language Names	2-6
Conventions Used in This Manual	2-7
Statement Descriptions	2-7
How to Read Syntax Diagrams	2-8
Notation Conventions	2-10
Extended BASIC Syntax	2-11
Standard Syntax	2-11
Extended Syntax /E Option	2-12
Statement Labels	2-12
Long Variable Names in XBASIC	2-13
Long Variable Names In Extended Syntax	2-14
Continuation Lines	2-15
Omitted Line Numbers /NL Option	2-17
Automatic Integer Conversion /I Option	2-18
Differences in Variables	2-19
Variable Arrays	2-19
ERR\$ System Variable	2-20
Modified Statements	2-21
CALL Statement	2-21
Parameters	2-22
Global Variable and Subroutine Names	2-22
COM Statement	2-24

CONTENTS, *continued*

DIM Statement	2-25
Three Dimensional Arrays ✈	2-25
Array Use in Subroutines	2-25
Conformal Dimensioning	2-26
Redimensioning Main Memory or Common Arrays	2-28
Redimensioning Virtual Arrays	2-28
FOR and NEXT Statements	2-29
ON-GOTO Statements	2-30
REM Statement	2-31
RESTORE Statement	2-31
STOP Statement	2-31
TRACE Statement	2-32
Branch Statements	2-32
Extended BASIC Statements	2-34
SUB Statement	2-35
SUBEND Statement	2-36
SUBRET Statement	2-37
ON-SUBRET Statement	2-38
Control Flow Statements	2-40
Extended IF Statement	2-40
LEAVE Statement	2-42
LOOP Statement	2-43
REPEAT Statement	2-44
SELECT Statement	2-45
WHILE Statement	2-48
Statements Unique To Extended BASIC	2-49
EXPORT Statement	2-49
IMPORT Statement	2-51
Unused Interpreted BASIC Instructions	2-52

INTRODUCTION

Extended BASIC, Compiled BASIC and Interpreted BASIC are similar forms of the BASIC language. Many descriptions in the Fluke BASIC Manual apply to all three forms. The Reference volume of the BASIC manual set covers Extended, Compiled, and Interpreted BASIC. In the Reference volume, language elements that are unique to XBASIC are marked



in the upper right corner of the page.

A Quick Reference Card, included in the manual set, contains a summary of the functions of Interpreted, Compiled and Extended BASIC. Use the reference card along with the Fluke BASIC Reference manual.

Section 2 of the Extended BASIC Manual introduces the Extended BASIC Language and explains Extended BASIC Syntax and programming statements. This section redefines statements that are similar to Interpreted BASIC but function differently, describes statements shared with Compiled BASIC, and describes statements unique to Extended Basic. The final pages list Interpreted BASIC statements that are not used in Extended BASIC.

Extended BASIC has some error codes that are different from Interpreted BASIC. These are listed in Section 6.

Unlike Interpreted BASIC, which uses only the interpreter program, Extended BASIC programs are developed using several programs: an editor, a compiler, and several linkage utility programs. Other sections of this manual describe the execution of Extended BASIC on the Instrument Controller, the program development process, and the linkage utility programs.

The Extended Basic Language

Extended BASIC is a compiled language, derived from Fluke Compiled BASIC. Extended BASIC utilizes the extended memory available in the Fluke 1722A Instrument Controller. Programs developed in Extended BASIC can occupy almost the entire four megabytes of memory that can be addressed by the TMS-99000 processor.

The extended version of BASIC programming language uses the BASIC compiler to translate a program into machine-executable code. The compiled version of the BASIC programming language, used in both Extended and Compiled BASIC, is essentially the same as the interpreted version. The key difference between compiled and interpreted languages is:

- The BASIC interpreter program translates a BASIC program into machine-executable code one line at a time while the program is in progress.
- The BASIC compiler program translates a BASIC program all at once, before the program is run. When the translated program is run, it is actually executed by the Runtime System Program.

The advantage of a compiled program is that program execution is not slowed waiting for translation at each step. A compiled program operates much faster than an interpreted one. This speed may make a critical difference in applications that cannot tolerate the processing delays of the Interpreted BASIC language.

The use of extended memory causes Extended Basic to be somewhat slower than Compiled Basic. Extended BASIC is useful in applications requiring very large programs where speed is not a primary consideration.

Because of the overhead involved in executing a program from extended memory, Extended BASIC is not as fast as Fluke Compiled BASIC; but it is faster than Interpreted BASIC.

Compiled programs are more compact than interpreted programs. The compiler makes more efficient use of space because it can translate the entire program at one time. There is more memory space available, since the interpreter program does not have to be available at the same time as the program.

Extended BASIC programs have the ability to use true subroutines written in Extended and Compiled BASIC. However, Compiled BASIC subroutines must be recompiled by XBC, the Extended BASIC Compiler, before linking. True subroutines may be developed separately from the calling program, may have parameters exchanged between them and the calling routine, and may have local variables that are not accessible from other program segments. True subroutines may be maintained in libraries for general use. True subroutines are defined by SUB and SUBEND statements that bracket them. Interpreted BASIC can use true subroutines, but only those written in another language such as FORTRAN or 99000 Assembly, not BASIC. Extended BASIC can use local subroutines like normal BASIC, or true subroutines written in FORTRAN, 99000 Assembly, Compiled, or Extended BASIC. Unlike CBASIC subroutines, XBASIC subroutines cannot be called from FORTRAN or Assembly programs.

Extended BASIC improves readability by allowing extended program line syntax and long variable names. Since the BASIC compiler program does not rely on line numbers, Extended programs may be written using descriptive labels instead of line numbers as targets for branch instructions. The BASIC compiler also allows source program lines to be continued on more than one display line. This permits the programmer to impart a visual structure to the program for better comprehension. Last, the interpreter program's two-character restriction on variable names is lifted, allowing descriptive variable names to be used.

Extended BASIC shares several capabilities with Compiled BASIC that are not directly related to the compiled nature of the language. Statements shared with Compiled BASIC simplify writing program segments involving loops and conditional branching. New statements in Extended BASIC (IMPORT and EXPORT) declare and use global variables.

Extended BASIC statements and syntax (both standard and extended syntax) are all extensions to ANSI Standard BASIC. Extended BASIC is a true superset of Compiled BASIC, except in one minor respect. Extended BASIC works exactly like CBASIC, but it allows longer XBASIC subroutine names in CALL statements.

Program Development with Extended BASIC

Extended BASIC programs are developed using several utility programs and run with a separate Runtime System Program. An editor program or the editor mode in the BASIC interpreter is used to create and modify the source program. A compiler program translates the source code into an object code file. A linking loader program is used to construct an executable file by combining object code files and subroutines and configuring them for FDOS. The BASIC Runtime System program is loaded into memory and run with the program that results from compiling the source code. All of these programs are included in the Extended BASIC package.

Language Names

There are three versions of the BASIC programming language available for Fluke Instrument Controllers. The names of these versions as used in this manual are:

Interpreted BASIC

Also referred to as BASIC, Fluke Enhanced BASIC, or IBASIC. This is the standard version of the BASIC language. It is the closest version to standard ANSI BASIC.

Compiled BASIC

Also referred to as CBASIC. This is the compiled version of the BASIC language. It is faster than Interpreted BASIC, and uses less memory.

Extended BASIC

Also referred to as XBASIC. It is the language described in this manual. XBASIC is similar to Compiled Basic.

CONVENTIONS USED IN THIS MANUAL

The following paragraphs describe the conventions used in this manual. Subsequent sections may have conventions of their own, which are defined on a per-section basis.

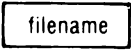
Statement Descriptions

Each Extended BASIC statement is described in a standard format.

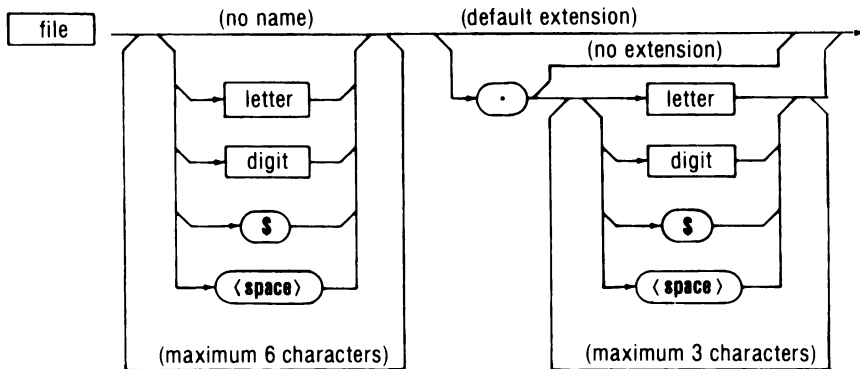
- Each statement is described in this manual and in the Reference volume of the IBASIC/CBASIC/XBASIC manual set.
- The statement description in this manual is an abridged description. The purpose of the statement is described and an example of the command-line syntax is given.
- The statement description in the Reference volume is the complete description. This description contains the syntax diagram, an example of the command-line syntax, a complete description of the statement, and in most cases, some programming examples.

How to Read Syntax Diagrams

A syntax diagram is a graphic representation of the construction of a valid command or statement in a programming language. The syntax diagram is a shorthand way of writing down all the rules for using the elements of a language. Since syntax diagrams are used throughout this manual, learning to read them can be a great time saver.

- | | |
|---|---|
|  | Words inside ovals must be entered exactly as they are shown. |
|  | Words inside boxes with rounded corners indicate a single key to be pressed, such as RETURN or ESC. |
|  | This indicates a space in the statement. (Press the spacebar.) |
|  | To create a control character, hold down the control key (CTRL), then press the other key. This one is a Control C; it causes a break in the program. |
|  | A box with lower case words inside means that you supply some information. In this case, you would enter a file name. |
| (explanation) | Words in parentheses are explanations of some kind. They give added information about the nearest block or path. |

From the left, any path that goes in the direction of the arrows is a legitimate sequence for the parts of a statement. This sample shows the correct syntax for naming a file. The translation is given below.



- A line exits the top of this diagram with no keyboard input. This indicates that it is possible to not specify the file name or its extension. In this case, the file would have no name, and the system would assign a default extension.
- Further down the diagram, you can see that there are other possibilities. You can choose up to six characters for the file name, moving once through the loop for each character chosen. Up to three characters can be chosen for the extension.
- File name and extension characters can be any combination of letters, digits, the \$ sign, and spaces.
- The file name and extension must be separated by a period as shown in the oval block at the top center.
- The remark “no extension” means that it is not necessary to specify an extension, even though a file name is given. Notice, however, that this remark occurs after the period, so the period is necessary if a name is specified.
- Here are some examples of valid file names according to the syntax illustrated in the diagram:

TESTIN.\$3A 1722A.RAC \$\$\$\$\$\$.\$\$\$

Notation Conventions

The conventions listed here are used for illustrating keyboard entries and to differentiate these entries from surrounding text. The braces, { }; brackets, []; and angle brackets, < > are not part of the keystroke sequence, but are used to separate parts of the sequence. Do not type these symbols.

- | | |
|----------------------|---|
| <xxx> | Means “press the xxx key”.
Example: <RETURN> indicates the RETURN key. |
| <xxx>/y | Means “hold down key xxx and then press y”.
Example: <CTRL>/C means to hold down the key labeled CTRL, and then press the key labeled C. |
| [xxx] | Indicates an optional input. Example: [input filename] means to type the name of the input file name if desired. If no file name is input, a default name will be used. |
| xxx | Means to type the name of the input as shown.
Example: BASIC means to type the program name BASIC as shown. |
| {xxx} | Indicates a required user-defined input.
Example: {device} means to type a device name of your choice, as in MF0: for floppy disk drive 0. |
| (xxx) | This construction has two uses: <ol style="list-style-type: none">1. As a separate word, (xxx) means that xxx is printed by the program. Example: (date) means that the program prints today’s date at this point.2. Attached to a procedure or function name, (xxx) means that xxx is a required input of your choice; the parentheses are a required part of the input. Example: TIME(parameter) means that a procedure specification is the literal name TIME followed by a parameter that must be enclosed in parentheses. |

EXTENDED BASIC SYNTAX

Extended BASIC programs may use the same syntax as interpreted BASIC programs (line number plus program statements). In addition, there are two syntactic options available: Standard Syntax and Extended Syntax.

- Standard Syntax with the BASIC compiler allows the use of program labels or line numbers as branch instruction targets.
- The Extended Syntax command-line option allows continuation lines and long variable names.
- The Extended Syntax option allows a free-form program layout that improves readability. Program line numbers may also be omitted when the /NL command-line option is used.

Using any of the Extended Syntax features requires a command-line option string when compiling. All of the command-line options are described in Section 3 of this manual. No options are needed to accommodate the additional Standard Syntax. Standard Syntax and Extended Syntax are described below.

Standard Syntax

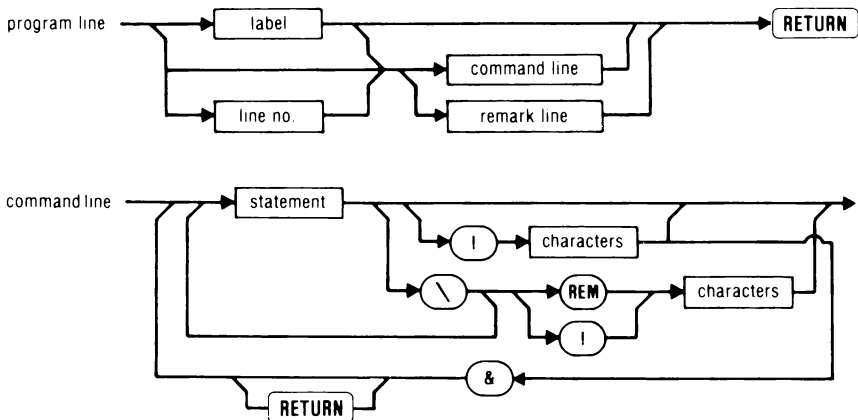
Standard Syntax allows the standard BASIC syntax as used in the interpreted version of BASIC. If you have a program that runs using the BASIC interpreter, the Standard Syntax is all that you need to compile the program.

- Command-line options are not needed to handle this syntax feature.

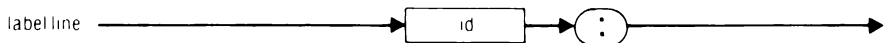
Extended Syntax /E Option

The Extended Syntax feature frees the programmer from the restrictions of single lines and two-character variable names of conventional BASIC.

The /E command-line option must be used when compiling programs using Extended Syntax (long variable names and continued lines). See Section 3 of this manual for additional information on the various command-line options.



Statement Labels



A statement label is a name that identifies an Extended BASIC program line. The label may be used in place of a line number in any branch instruction. (See Branch Statements in this section.)

- A statement label consists of any legal variable name, followed by a colon character (:).
- A statement label may only appear at the beginning of a statement.
- A statement label may appear as the only component on a line.
- These are all legal statement labels:

a:
a1:
begin:
dvm:

Long Variable Names in XBASIC

The Extended BASIC language allows variable names of any length, unlike Interpreted BASIC, which allows only two. Longer, more descriptive variable names improve the readability of the program.

- The first eight characters of a name must be unique. For example:

CONTROLLOCK
CONTROLLER

are considered to be the same variable.

- The case of all variable names is ignored. For example, these names are all be considered to be the same:

NAME
name
NaMe
naME

- Variable names may also include an underline character to improve readability. These variable names have been clarified with the addition of underline characters:

PAPER__FEED
OHMS__ADJUST
TOTAL__INPUT__VALUE

Long Variable Names In Extended Syntax

The characteristics of variable names change when Extended Syntax is used.

- When long names are used, spaces, tabs, parenthesis, or arithmetic operators must be used around all keywords, variable names, subroutine names, and numbers. For example,

```
IFCOUNTER=SCORETHENMARKER=MARKER+1
```

won't work if Extended Syntax is used. It is impossible for the compiler to determine what to do. The example statement needs to be clarified by adding spaces:

```
IF COUNTER=SCORE THEN MARKER=MARKER+1
```

- The Extended Syntax allows variable names to begin with keywords.
- Variable names may not begin with the letters FN, even though the Extended Syntax is used. The BASIC Compiler program will interpret all strings that begin with FN as user-defined functions.

Continuation Lines

In Extended BASIC, program structure is not tied closely to line numbers. As a result, statements may be longer than the width of the Controller's screen. The Extended Syntax option allows a programmer to insert <RETURN> characters into a program line to improve readability without having those <RETURN> characters signify the end of the line. <RETURN> characters used for line continuation are preceded by an ampersand (&) character.

- The embedded <RETURN> characters are required if the BASIC source program is going to be printed, manipulated using the File Utility program, or used with other facilities with a fixed line length that is shorter than the Extended BASIC line.

For example, trying to print a 150-character command line on an 80 column printer will cause the printer to print the first 80 characters of the line over to the right margin, then print the remaining 70 characters on top of each other in the last column. Imbedding <RETURN> characters in the text (using the & character) will allow the printer to print the entire program line on more than one printed line.

- Aside from its function as a continuation marker, an ampersand is functionally equivalent to a space in the program line.
- All statements may be continued.
- A single language element, (a keyword, variable name, string, or numeric constant) may not be continued.

For example:

```
100 PRINT "A long string like this one may NOT &  
be continued to a new line"
```

The example program line is illegal because the string is broken in mid-line. You can overcome this problem by breaking the string in two and concatenating the string.

```
100 PRINT "A long string like this one may" &  
+ "be continued to a new line like this."
```

Extended Basic Syntax

- The compiler will support a minimum of 24 continued lines (80 character line). The total number of continued lines could be as many as 200 per initial line, depending on the amount of memory available.
- The maximum length of a continued line is 2000 characters. A line can be continued indefinitely until the maximum character count is reached.
- A continued line, no matter how many times it is extended to a new display line, is considered one line for the purposes of error reporting.
- Remarks (identified by the ! character only) may be placed anywhere in a continued command line that a space character is legal, and terminated with a continuation character (&). This allows remarks to be placed into continued lines as shown below:

```
IF PAGE LENGTH < LINE_COUNT THEN    ! IF AT END OF PAGE      &  
    PRINT FORMFEED$                  !    SKIP TO NEXT PAGE &  
!ENDIF
```

- The BASIC Compiler program does not require the imbedded <RETURN> characters. Only line-length limited programs such as the File Utility program, and peripherals such as line printers, require imbedded <RETURN> characters.

For example, the Extended Syntax option (with long variable names and continued lines) allows the programmer to create a single Extended BASIC command line that looks like this:

```
100 !BEGIN CAP COUNTER PROCEDURE  
110     IF BOTTLE COUNTER > CAP COUNTER &  
        GOTO 200 ! THE LABEL CHECKING PROCEDURE &  
        ELSE &  
            CALL RESET ! TO RECYCLE THE BOTTLE CAP SEQUENCE &  
        !END IF &  
!END PROCEDURE
```

Omitted Line Numbers /NL Option

An Extended BASIC programmer has the option of using line numbers on every statement, as in Interpreted BASIC, or using line numbers only as needed for statement labels.

The /NL command-line option specifies to the BASIC Compiler program that line numbers are not used. Command-line options are discussed in Section 3 of this manual.

- If the /NL option is not used, a line number must appear at the start of every non-continued line.
- Line numbers (if used) must be in increasing numerical order.
- All line numbers in a single source program file must occur in increasing numerical order.
- Line numbers may not be duplicated within a single source program (including true subroutines which are part of the single source file).
- There are no restrictions to duplicating line numbers in true subroutines that are compiled separately.

With the /NL option, the following code segment is legal.

```
START:
  IF A=12 THEN &
    GOTO FINISH &
  ELSE &
    GOTO START &
FINISH:
```

If line numbers are not used, system messages will refer to lines in sequential order from the beginning of the program. For example, a message referring to an error in line 13 will refer to the thirteenth line from the top of the program, not a line numbered 13.

Automatic Integer Conversion /I Option

The BASIC Compiler program can convert eligible constants into integers, whether or not a % character follows the number. Bypassing the % character saves program space and reduces the programmer's typing time. Integer operations typically execute more quickly than floating-point operations, so a speed advantage is realized, too.

The /I command-line option invokes the automatic integer feature of the BASIC compiler. The command-line options are described in Section 3 of this manual.

- Any integer will be stored as an integer, with or without a % character following it.
- A number may be exempted from conversion to an integer by including a decimal point or an exponent field.

For example, these two numbers will be stored as integers:

5
312

To reserve these two numbers as floating-point numbers, include a decimal point or an exponent:

5.
312E0

DIFFERENCES IN VARIABLES

Extended BASIC is similar to Compiled Basic in the treatment of variables. Both differ from Interpreted Basic in the expansion of Variable Arrays to three dimensions, and in incorporating the system variable, ERR\$. In addition, if Extended Syntax is used, a variable name may consist of more than two characters.

Variable Arrays

In Extended BASIC, variable arrays may have three dimensions. For example, it is possible to have an array element

A(2,9,6)

Other specifications that apply to dimensioned arrays in Interpreted BASIC (with the exception of name length) apply as before.

For the syntax of the DIM statement associated with three-dimensional arrays, see the DIM Statement elsewhere in this section and in the Reference volume.

ERR\$ System Variable

The ERR\$ System Variable contains a string that is the name of a module where an error occurred. This is a companion to the ERR System Variable that exists in IBASIC, which contains the error number. For subroutines, the name that ERR\$ will contain is the one specified by the SUB statement. (Refer to The SUB Statement elsewhere in this section.) For the Main program segment, the name will be \$MAIN\$.

For example, suppose an arithmetic overflow occurs in a subroutine named SCORE. Checking the ERR and ERR\$ variables, shows that they contain

```
ERR=601 (Arithmetic Overflow)  
ERR$=SCORE
```

This variable is useful in program development and in error handling subroutines. It allows the programmer to determine the source of an error. When properly written, the error handler may output its own error messages, perhaps tailored to the exact experience level of the end user of the program.

MODIFIED STATEMENTS

The statements described below are similar to Interpreted BASIC statements, but operate slightly differently in Extended BASIC programs. Only the differences between Extended and Interpreted, or Extended and Compiled statements are described.

Some statements in Interpreted BASIC operate in both the Immediate Mode and in the Run Mode. In Extended and Compiled BASIC, these statements only operate in the Run Mode. If this is the only difference, the statements are not described here.

Use these statement descriptions to supplement the descriptions that are in the Fluke BASIC Programming Manual. The statements are described alphabetically.

CALL Statement

The CALL Statement works slightly differently in IBASIC, CBASIC and XBASIC.

With CBASIC:

- The number of parameters exchanged with the calling routine is not limited to 10.
- Subroutine names may be of any length. In CBASIC, only the first six characters of the subroutine name are significant.

With XBASIC:

- Subroutine names may still be of any length, but different types of subroutines recognize different numbers of characters as significant.
- For calls to Assembly or FORTRAN subroutines, only the first six characters of the subroutine name are significant. For calls to XBASIC subroutines, the first eight characters are significant.

Parameters

There is no limit to the number of parameters that may be exchanged between a true subroutine (a subroutine bracketed by SUB and SUBEND statements) and its calling routine.

Global Variable and Subroutine Names

In Extended BASIC, as in Compiled BASIC, variable and subroutine names may be of any length, but are limited in the number of significant characters. Global variable names (used in EXPORT and IMPORT statements) are significant to 8 characters. All names are converted to upper case in the output file.

BASIC subroutine names are also significant to 8 characters and are converted to upper case in the .OBX file.

BASIC CALL statements that refer to machine code subroutines should use names that are not more than 6 characters long. A long name in the .OBX file, generated by the Extended Basic Compiler, will not be matched by the Extended Library Linker Program with the shorter name in the .OBJ or .LIB file. (See Section 3 for a discussion of file types.)

Note that even though longer subroutine names are allowed to improve readability, only the first six or eight characters are significant in subroutine names.

For example, the two names

CONTROLLER
CONTROLLOCK

appear to be different, although the first six characters are the same. If they are calls to Assembly or FORTRAN subroutines, the first six characters are significant, and they will be considered the same subroutine. If they are calls to XBASIC subroutines, or global variable names used in EXPORT and IMPORT statements, the first eight characters will be significant, and two different subroutines will be called.

To simplify programming, use the following rules when Assembly or FORTRAN routines are called:

- Remember that Extended BASIC converts subroutine names to upper case. This means that an Assembly program should use upper case labels for entry points that may be called from BASIC. The FORTRAN Compiler always generates upper case labels for subroutines, so this should not be a problem.
- If a CALL statement refers to a FORTRAN or Assembly routine, remember that the subroutine name used in the CALL statement should be less than or equal to 6 characters long. A long name in the .OBX file, generated by the Extended Basic Compiler, will not be matched by the Extended Library Linker Program with the shorter name in the .OBJ or .LIB file.

COM Statement

The COM statement is used for sharing variables between chained programs. In addition, Extended BASIC programs also use the COM statement to make certain variables accessible to both calling routines and true subroutines.

- The COM statement must be used first in the Main program segment to assign variables in common. After that, any subroutine can access these variables in common. For example, in the Main program, this statement

```
COM A, B%(40)
```

assigns the two items: the real variable A, and the array B%, with 41 elements in common to all program sections. Later, a subroutine may use all or some of these variables with another COM statement. This statement

```
COM Q, G%(40)
```

in a subroutine specifies that the real variable and array from the Main program segment may also be used in this subroutine.

Note that the actual names used in the subroutine COM statement are not significant. In the example above, the subroutine's variable Q is the same as A in the Main program.

- All COM statements (whether in a main program, a subroutine, or a chained program) must allocate the same amount of space (the same number of variables of the same type).
- No strings may be used with the COM statement.

DIM Statement

In Extended BASIC, variable arrays may be dimensioned to three dimensions, and arrays may be exchanged by reference between calling routines and true subroutines.

Three Dimensional Arrays

The DIM Statement may be used to dimension arrays up to three dimensions, as opposed to two dimensions in Interpreted BASIC. (Refer to the earlier discussion of Variable Arrays for more information.)

For example, this statement:

```
DIM A(5,4,8)
```

defines a three-dimensional array of six elements (0 through 5), with each element containing five sub-elements (0 through 4), and each sub-element containing nine sub-sub-elements (0 through 8). Refer to Appendix A for the supplementary syntax diagram for dimensions.

Array Use in Subroutines

The DIM statement is changed to allow subroutines to use arrays that are passed by reference from the calling routine. There are two ways in which this is accomplished, conformal dimensioning and redimensioning.

Conformal Dimensioning

Conformal dimensioning is the recommended method for passing arrays by reference to subroutines. The following program segment is an example of conformal dimensioning:

```
100 SUB ABC(K%(), I%)  
110 DIM K%()  
120 ...
```

Line 100 defines a true subroutine named ABC with two parameters exchanged between it and its calling routine (see SUB Statement). The DIM statement in line 110 specifies that the variable array K%() will have the same number of dimensions (subscripts) and the same dimension limits as it does in the calling routine.

- The variable array may be a virtual array or a normal array in Main memory or Common memory.
- The number of dimensions (subscripts) used in the subroutine must match the dimensions in the array. For example, if K%() is a two-dimensional array, but the subroutine tries to use it as a one-dimensional array, an error will result.
- The dimension limits of the array will be the same in the subroutine as in the calling routine. In the above example, if the original dimension on K%() was

DIM K%(10)

in the calling routine, an attempt to use array element K%(11), will cause a Subscript Out of Range error.

- The data type of the original array will be used in the subroutine. In the example, K%() is an array of integer variables. An attempt to use this array for another data type (floating-point for instance) will result in an error.

- Conformal dimensioning may be used across more than one level of subroutines. For example, a Main program segment calls Subroutine A, which in turn calls Subroutine B. A DIM statement in Subroutine B may be used to access an array that was originally dimensioned in the Main program segment. The array does not have to be used in Subroutine A in order to be used in Subroutine B.
- An array may be originally dimensioned in a subroutine, then dimensioned conformally in subsequent subroutines that are called from it. The original DIM statement does not have to be in the Main program segment.

Redimensioning Main Memory or Common Arrays

The second method for dimensioning a variable array in a subroutine is to redefine the array dimensions in the subroutine. For example, in the following statements

```
100 SUB ABC(KZ(), IX)  
110 DIM KZ(100)  
120 ...
```

Line 100 defines a true subroutine named ABC with two parameters exchanged between it and its calling routine (see SUB Statement). The DIM statement in line 110 dimensions array K%() to a 101-element single-dimension array.

- The new dimensions for the array only apply within this subroutine.
- Virtual arrays may not be redimensioned with this method. An Illegal Parameter DIM error (error 908) will occur if an attempt is made to redimension a virtual array.

Redimensioning Virtual Arrays

If a subroutine uses a virtual array element, but conformal dimensioning is not desired, the following method may be used:

```
100 SUB ABC(KZ(), IX, JX)  
110 DIM #IX, KZ(JX)  
120 ...
```

Line 100 defines a true subroutine named ABC, which exchanges three items with its calling routine (see SUB Statement). The DIM statement in line 110 dimensions the array K%() and assigns a channel number by means of the values passed from the calling routine.

- An Illegal Parameter DIM error (error 908) will occur if the parameter is not a virtual array or if the array is attached to a channel other than the one specified in the DIM statement.

FOR and NEXT Statements

FOR and NEXT Statements operate essentially the same in IBASIC, CBASIC AND XBASIC programs. Compiled and Extended BASIC differ from Interpreted BASIC in two ways:

- Only one NEXT statement is permitted for each FOR statement. Multiple NEXT statements in a FOR/NEXT loop are not permitted.
- Exits via statements other than NEXT statements (GOTO for instance) are permitted.

ON-GOTO Statements

ON-GOTO interrupt processing statements in Extended BASIC subroutines may be nested in different subroutine levels. ON-GOTO interrupts that have been nested within other subroutines become local to that subroutine. This means that they are only active while the subroutine that contains the ON-GOTO statement is processing.

In the following program fragment, the ON CTRL/C statement in the MAIN program segment sets up a CTRL/C handler named TRAP. Any CTRL/C interrupts received during the execution of this portion of the program will branch to the handler at TRAP.

Within the MAIN segment, a subroutine call is made to subroutine SUBL. This subroutine sets up a local CTRL/C handler called SUB_TRAP. Any CTRL/C interrupts received during this portion of the program will branch to the local CTRL/C handler, SUB_TRAP. Once program execution returns to the MAIN segment, the CTRL/C handler called TRAP is active once more.

```
MAIN:
  ON CTRL/C GOTO TRAP
  .
  .
  CALL SUBL(AZ,BZ)
  .
  .
  STOP
TRAP:
  ! ctrl/c handler for main
  .
  .
  RESUME

SUB SUBL (AZ,BZ)
  ON CTRL/C GOTO SUB_TRAP !goto local ctrl/c handler
  .
  .
  SUBRET
SUB_TRAP:
  ! local ctrl/c handler
  .
  .
  RESUME
SUBEND
```

REM Statement

The ! form of the REM statement may be embedded into continued lines for better readability. The Extended Syntax option allows a programmer to insert <RETURN> characters into a program line to improve readability without having those <RETURN> characters signify the end of the line. <RETURN> characters used for line continuation are preceded by an ampersand (&) character.

- Remarks (identified by the ! character only) may be placed anywhere in a continued command line that a space character is legal, and terminated with a continuation character (&). This allows remarks to be placed into continued lines as shown below:

```
IF PAGE LENGTH < LINE_COUNT THEN    ! IF AT END OF PAGE      &  
    PRINT FORMFEED$                  !   SKIP TO NEXT PAGE  &  
!ENDIF
```

RESTORE Statement

The RESTORE statement resets the pointer to a data item that will be read more than once in a program. It is used in conjunction with the Read and Data statements. In Extended BASIC, the RESTORE statement may refer to a labeled line as well as to a line number. See the descriptions under Branch Statements and Statement Labels elsewhere in this section.

STOP Statement

Only the simple form of the STOP statement is allowed in Extended BASIC. The STOP ON (line number) form of the STOP statement is used only for interactive debugging in Interpreted BASIC programs.

TRACE Statement

The only form of the TRACE statement used in Extended BASIC programs lists line numbers and subroutine names. Traces of variables and trace results sent to a channel are not permitted in compiled programs. The form of the TRACE statement that specifies a beginning line number is intended for interactive debugging of BASIC programs and is not used in Extended BASIC.

Here is an example of a trace in a Main program segment, showing line numbers and subroutine names.

```
$MAIN$  
@130  
@145  
@150  
SUB1  
@10  
@30
```

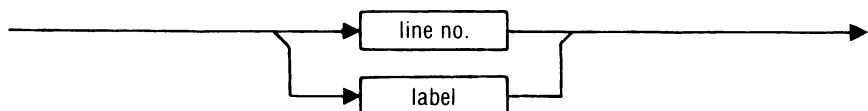
Branch Statements

In Interpreted BASIC, all of the BASIC statements that involve a branch of control may branch to a line number. In Extended BASIC, they may also branch to a labeled line. The statements that branch are:

```
GOTO  
ON-GOTO  
IF-GOTO  
IF-THEN GOTO  
IF-THEN GOTO-ELSE GOTO  
RESTORE  
RESUME
```

Note that, while RESUME is not a branch statement, it is included here for consistency. Its use of line references is similar to a branch instruction.

The following syntax diagram shows the target that may be inserted into any of the syntax diagrams for the statements listed above.



- The label must be defined in the same program unit (i.e., in the main program or a subroutine) in which it is used.
- GOTOs must be included in IF-THEN and IF-THEN-ELSE statements to distinguish a label from a subroutine name. For example, the statement

```
100 IF V=0 THEN MEASURE ELSE DISPLAY
```

specifies that a subroutine named MEASURE will be called implicitly if V=0, and a subroutine named DISPLAY will be called if V does not equal 0. To make these into branches to labels, insert GOTOs into the same statement:

```
100 IF V=0 THEN GOTO MEASURE ELSE GOTO DISPLAY
```

Now the statement will branch to a line labeled MEASURE if V=0; otherwise, the statement will branch to a line labeled DISPLAY.

For a description of statement labels, see Statement Labels earlier in this section.

EXTENDED BASIC STATEMENTS

The following statements are unique to the compiled versions of the BASIC language. These statements provide the capability to use true subroutines and to improve program control flow. A complete description of each statement is included below.

The Extended BASIC subroutine statements are used to create true subroutines in BASIC. Unlike Interpreted BASIC, which only uses true subroutines written in a language other than BASIC (FORTRAN, Assembly, etc.), Extended BASIC programs may also use true subroutines written in Compiled and Extended BASIC.

SUB Statement

Usage: SUB {subroutine name}(parameter list)

The SUB statement defines a true Extended BASIC subroutine. All statements following the SUB statement (until terminated by a SUBEND statement) constitute an Extended BASIC subroutine. The SUB statement also defines a list of parameters that will be passed to that subroutine from the calling routine.

- The SUB statement must be the first statement on a program line.
- Parameters may be passed from a calling routine by value (as an expression) or by reference (by name, as a variable or an entire array name).

An example of a SUB statement is

```
100 SUB DVM(READINGS$, MESSAGE$, SCALE)
```

which identifies the beginning of a subroutine named DVM that will exchange three parameters with a calling routine. The parameters are an integer array READINGS\$(), a string called MESSAGE\$, and a floating-point parameter named SCALE. All of the statements between this SUB statement and the next SUBEND statement are part of the subroutine DVM.

This sample subroutine will be used via a CALL statement like this:

```
100 CALL DVM(AZ(), B$, C)
```

When the subroutine DVM runs, it will use the variable array A%() from the calling routine, with the local name of READINGS%(). B\$ From the calling routine will be used in the subroutine as MESSAGE\$, and C will be used as SCALE.

SUBEND Statement

Usage: SUBEND

The SUBEND statement marks the end of a true subroutine. All of the statements occurring between this statement and the last SUB statement constitute a subroutine.

- Only one SUBEND statement may be used with each SUB statement.
- The SUBEND statement returns program control to the routine that called the subroutine.

SUBRET Statement

Usage: SUBRET

The SUBRET statement returns control from a subroutine back to the calling routine. SUBRET is used to return to a calling routine from places other than at the end of a subroutine.

- Any number of SUBRET statements may be used in a subroutine.
- The SUBRET statement must be used within the body of a subroutine.
- The SUBRET statement is analogous to the RETURN statement in BASIC subroutines that are entered with a GOSUB statement.

ON-SUBRET Statement

Usage: ON [condition] SUBRET

The SUBRET statement may be used as an instruction for processing an interrupt that occurs while a program is executing a subroutine. When an interrupt occurs, the SUBRET statement causes control to transfer from the subroutine back to its calling routine. The interrupt will still be in effect when control returns, which allows interrupt processing via an ON-GOTO statement in the calling routine.

- Any of the available interrupts (ERROR, CTRL/C, etc) may be handled by an ON-SUBRET statement.
- Any number of ON-SUBRET statements may be cascaded before reaching the point where interrupt processing occurs. This allows several layers of subroutines to return to a main program (if desired) before processing interrupts.

For example, a Main program segment calls subroutine A, which in turn calls subroutine B, which in turn calls subroutine C. An interrupt occurs while subroutine C is in progress. ON-SUBRET statements in subroutines A, B, and C would send control all the way back to the Main segment in turn, where interrupt processing can be performed.

```
!MAIN
  ON CTRL/C GOTO TRAP
  .
  .
CALL A
  .
  .
END

SUB A
  ON CTRL/C SUBRET
  CALL B
SUBEND

SUB B
  ON CTRL/C SUBRET
  .
  .
  CALL C
SUBEND

SUB C
  ON CTRL/C SUBRET
  .
```

Note that, without the ON-SUBRET statement, an interrupt that occurs in a subroutine will be processed by an interrupt handling statement in the calling routine (if one exists), after which control is returned to the subroutine at the spot where the interrupt occurred. After using an ON-SUBRET statement to respond to an interrupt, the subroutine where the interrupt occurred must be reentered with another CALL statement.

Control Flow Statements

Extended BASIC offers several means of altering the control of flow during program execution. In addition to the FOR-NEXT, IF-THEN-ELSE, GOSUB, and GOTO statements found in Interpreted BASIC, X BASIC provides several means of creating loops, exiting loops, and allowing conditional branching. Descriptions of these statements are found in the following paragraphs.

Extended IF Statement

```
Usage: IF [expression] THEN
        program statements
      ELSIF [expression] THEN
        program statements
      ELSIF [expression] THEN
        program statements
      ELSE [expression]
        program statements
      ENDIF
```

The IF statement provides the principal means of selecting between several alternative courses of action. For instance, consider the amount of code required by a set of nested IF-THEN-ELSE statements used to select one action based on a value.

```
10  if a = 0 then 999 else 20
20  if a = 1 then 100 else 30
30  if a = 2 then 200 else 40
40  if a = 3 then 300 else 50
:
:
:
999 close all \ exit
```

The Extended IF statement makes this easier by allowing the ELSE condition to be the next IF condition. For example,

```
begin:
  if a = 0 then
    call sub0 ()
  elseif a = 1 then
    call sub1 ()
  elseif a = 2 then
    call sub2 ()
  elseif a = 3 then
    call sub3 ()
  else
    print "selection not possible"
    print cpos(15,10); "Touch Screen to Continue"
    k% = key \ wait 00:01 for key
    goto begin
  endif
```

An Extended IF is distinguished by an IF statement that has nothing (except an optional trailing remark and the end of the line) following the THEN keyword. The entire Extended IF consists of the following elements:

- An “IF {expression} THEN” statement that starts the IF. The statements following the THEN are executed if the {expression} is true (nonzero).
- Zero or more “ELSIF {expression} THEN” statements that select one of several alternatives if the initial “IF {expression} THEN” clause was not selected. An ELSIF statement is executed if the {expression} is true (nonzero). The ELSIF statements are executed in the order in which they appear in the source program.
- An optional “ELSE” statement that is executed if none of the other alternatives were selected. Note that at most one ELSE may be used with any given Extended IF.
- A mandatory “ENDIF” statement that terminates the Extended IF.
- These Extended IF statements must be the only statements in a program line. This restriction eliminates any interaction with “simple IF” statements, which are still part of the Extended BASIC language.
- Extended IF statements may be nested to any desired depth; no confusion (at least where the compiler is concerned) is possible. We recommend that you indent the program text to make it easier to determine which code is under the control of which IF.

LEAVE Statement

Usage: **LEAVE**
 LEAVE IF {expression}

The **LEAVE** statement is used to leave a structured loop (**WHILE**, **REPEAT**, **LOOP**, or **FOR**). It has the two forms shown above.

- The first form (**LEAVE**) simply jumps out of the innermost loop unconditionally.
- The second form (**LEAVE IF**) leaves the loop only if the {expression} has a true (nonzero) value.
- The **LEAVE** statement may be subordinate to an **IF** or other statement; it simply causes an exit from the innermost loop in which it occurs.

The following program fragment illustrates the two constructions of the **LEAVE** statement.

```
LOOP
  PRINT "Enter a value";
  INPUT aZ
  IF (aZ < 0Z) THEN
    PRINT "Negative. Leaving the loop"
    LEAVE
  ELSIF (aZ <= 10Z) THEN
    PRINT "Between 0 and 10"
    LEAVE IF aZ = 5Z      ! leave for 5
  ELSE
    PRINT "Greater than 10."
  ENDIF
ENDLOOP
```

LOOP Statement

Usage: **LOOP** ! begin infinite loop

```
...  
... statements ...  
...  
ENDLOOP
```

The **LOOP** statement can be used to implement an infinite loop.

- The **LOOP** statement has no explicit termination test; it normally uses the **LEAVE** statement to provide a loop exit.
- Control stays within the **LOOP** - **ENDLOOP** keywords until a **LEAVE** statement, **GOTO** statement, or interrupt is encountered.

The following program fragment repeatedly asks for another value until a zero is entered from the keyboard. The **LEAVE** statement causes control to leave the loop when the value received is zero.

```
LOOP          PRINT "Enter another value";  
              INPUT a%  
              LEAVE IF a% = 0%  
              PRINT "Try again...."  
ENDLOOP
```

REPEAT Statement

Usage: REPEAT

```
...  
... statements ...  
...  
UNTIL { expression }
```

The REPEAT statement introduces a REPEAT - UNTIL loop. Unlike the WHILE loop a REPEAT - UNTIL loop makes its termination test at the bottom of the loop rather than at the top.

- A REPEAT - UNTIL loop is always executed at least once, since the termination test is not made until the bottom of the loop has been traversed.
- The action of this loop is to execute the statements in the loop body repeatedly until the value of the {expression} is true (nonzero).

The following loop reads lines of input from channel 2 until a line starting with a letter is found:

```
REPEAT  
    PRINT "Reading another line"  
    INPUT LINE #2, a$  
    ch$ = LEFT(LCASE$(a$), 1$)  
UNTIL "a" <= ch$ AND ch$ <= "z"
```

SELECT Statement

```
Usage: SELECT {selector expression}
      CASE {case expression}
      ...
      ... statements ...
      ...
      CASE {case expression}
      ...
      ... statements ...
      ...
      CASE ELSE      ! anything not caught above
      ...
      ... statements ...
      ...
      ENDSELECT
```

The SELECT statement implements an “n-way” branch, depending upon the value of a “selecting” expression. In other languages, the SELECT statement is sometimes known as a CASE statement.

- The case expression is a comma-separated list of values formatted as:

```
≤ expression
≥ expression
<> expression
= expression
< expression
> expression
expression
expression_1 .. expression_2
```

- The form:

CASE expression__1 .. expression__2

says that if the value of the {selector expression} falls between the values expression__1 and expression__2, that is:

expression__1 ≤ selector expression ≤ expression__2

then the corresponding CASE should be executed.

- The “selector” may be any data type including strings.
- The CASE clauses simply indicate the values of the {selector expression} for which the corresponding piece of code should be executed. The tests for the different cases are performed in the same order as their appearance in the program until one of the cases matches the value required. At this point, the code following the CASE is executed until the next CASE occurs, then control flows to the ENDSELECT statement; the remaining cases are not evaluated.
- Any of the CASE expressions may involve variables.
- An “illegal mode mixing” error is reported if the value of the selector and case values are not compatible (i.e., number compared with string).
- You can specify an empty CASE clause (a CASE without any statements following). This permits an explicit “do nothing” CASE to weed out selector values for which no action should be performed. This is illustrated in the following example.

As an example of the SELECT statement, consider the following program fragment:

```

while (len(string$) <= 5% and eof% = 0%)
  char% = getchar(channel%)
  select (char%)

    case 13%                ! carriage return
      leave                ! leave WHILE loop

    case 26%                ! EOF
      eof% = -1%          ! signal end of file

    case 32%, 9% .. 10%     ! space, tab, linefeed
      ! do nothing

    case < 32%, >= 127%     ! other control character
      print "Funny character"; char%: "encountered"

    case 48% .. 57%, 65% .. 90% ! digit or uppercase
      string$ = string$ + chr$(char%)

    case 97% .. 122%        ! lowercase letter
      string$ = string$ + ucase$(chr$(char%))

    case else                ! other punctuation
      punct$ = punct$ + chr$(char%)

  endselect

  !
  ! select course of action from short option list
  !
  select (string$ + punct$)

  case a$ + "!"

    print "Quick mode"

  case "end"
    leave

  case "quit"
    exit

  endselect
endwhile

```

WHILE Statement

Usage: WHILE {expression}

```
...  
... statements ...  
...  
ENDWHILE
```

The WHILE statement introduces a WHILE loop. The {expression} is rounded to an integer and evaluated. While the expression's value is true (nonzero), the loop surrounded by the keywords WHILE and ENDWHILE is executed. For example:

```
WHILE iZ < 105% OR iZ > 240%  
    CALL getval(iZ)  
    PRINT "The new value is"; iZ  
ENDWHILE
```

- The {expression} is evaluated at the top of the loop, thus the loop is not executed if the {expression} is evaluated as false.

STATEMENTS UNIQUE TO EXTENDED BASIC

EXPORT and IMPORT, statements unique to the Fluke extended version of the BASIC language, declare and use global variables. Global variables allow global access to variables between programs and subroutines. EXPORT defines and reserves memory space for a global variable. IMPORT references the address of a global variable for compiler access to variables in another program module.

EXPORT Statement

Usage: EXPORT {variable list }

Unlike the COM statement, which is used to communicate between different programs, the EXPORT statement is used to communicate between subprograms that are part of one executable program. EXPORT is used to declare a set of variables and arrays as global variables, assigning memory space to the variables and arrays in the variable list and publishing the variable names in the output .OBX file as global variable definitions.

- Variables and arrays in the variable list may be integer, floating-point, or string variables.
- Arrays must be dimensioned so that the Extended Basic compiler can notify the Extended Linking Loader to reserve storage space for them. Dimension values must be constant integers or integer-valued real numbers.

- Global variables are always preset to zero (for numbers) or the null string (for strings) when the program begins execution.

A program module is a BASIC main program or a true subroutine. The following rules govern variables in program modules:

- A variable may not be named in both an EXPORT statement and a DIM, COM, IMPORT, or another EXPORT statement in the same program module.
- Any number of EXPORT statements may appear in a single program module.
- A global variable should be declared in an EXPORT statement before using it in another BASIC statement in the same program module.

The following program segment illustrates the use of the EXPORT statement:

```
10 sub get_readings(howMany%)
20 export dvmReadings(100)
30 import dvmAddress%, dvmGetData$
40 print @dvmAddress%, dvmGetData$
50 for i% = 1% to howMany%
60 input @dvmAddress%, dvmReadings(i%)
70 next i%
80 subend
```

Line 20 defines dvmReadings as a global variable (array) and reserves memory space for 100 elements. The variable dvmReadings may now be accessed from other programs and subroutines. (The IMPORT statement accesses the global variable, dvmAddress% from another program or subroutine where it was defined with an EXPORT statement.)

IMPORT Statement

Usage: `IMPORT {variable list}`

The `IMPORT` statement is used to declare a set of variables, which have been defined in a program module (BASIC main program or a true subroutine), as global variables. `IMPORT` publishes the variable names in the output `.OBX` file as global variable references.

- Variables and arrays in the variable list may be integer, floating-point, or string variables.
- An array declared in an `IMPORT` statement should use only an empty pair of parentheses “()” in the `IMPORT` statement to indicate that the variable is an array. No array subscripts may be used in the `IMPORT` statement itself.
- A variable may not be named in both an `IMPORT` statement and a `DIM`, `COM`, `EXPORT`, or another `IMPORT` statement in the same program module.
- Any number of `IMPORT` statements may appear in a single program module.
- A global variable should be declared in an `IMPORT` statement before using it in another BASIC statement in the same program module.

The following program segment illustrates the use of the `IMPORT` statement:

```
10 sub printStrings(start%, finish%, channel%)
20 import array$( )
30 for i% = start% to finish%
40 print #channel%, array$(i%)
50 next i%
60 subend
```

Line 20 in subprogram `printStrings` accesses the global variable `array$` from another subprogram or main program. The empty parentheses indicate that the variable is an array.

UNUSED INTERPRETED BASIC INSTRUCTIONS

Extended BASIC does not have the operating modes that Interpreted BASIC does. Of the four modes in Interpreted BASIC (the Immediate mode, the Edit mode, the Run mode, and the Step mode), Extended BASIC only retains the Run mode. Therefore, none of the Immediate mode, Edit mode, or Step mode commands are used in Extended BASIC. Attempting to use these commands with Extended BASIC will cause a syntax error. In Extended BASIC, editing functions are performed by a Text Editor utility program. Statements and commands not used in the extended version of the BASIC language are listed in Table 2-1.

Table 2-1. Unused Instructions

INSTRUCTION	DESCRIPTION
CONT	The CONTinue Command is a program debugging command that is used only in the Immediate mode. Extended BASIC does not have an Immediate mode.
DELETE	The DELETE Command is a editing command that is used only in the Immediate mode. Extended BASIC does not have an Immediate mode. Editing is done with an editor accessory program.
EDIT	The EDIT Command is used only in the Immediate mode. Extended BASIC does not have an Immediate mode. Editing is done with an editor accessory program.
LINK	The LINK Command is used to load external subroutines. External subroutines are linked in during the linking process, using the Linking Loader or Linkage Editor programs.
LIST	The LIST Command is used only in the Immediate mode. Extended BASIC does not have an Immediate mode.
OLD	The OLD Command is used only in the Immediate mode. Extended BASIC does not have an Immediate mode.
REN	The RENumber Command is an editing command that is used only in the Immediate mode. Extended BASIC does not have an Immediate mode.
RESAVE and RESAVEL SAVE and RESAVE	These statements are not used in Extended BASIC. Program editing and managing are done with a separate Editor accessory program.
STEP	The STEP Command is a program debugging command that is used only in the Immediate mode. Extended BASIC does not have an Immediate mode.
UNLINK	The UNLINK Command is used to delete external subroutines from memory in the interpreted version of BASIC.

Section 3

Extended BASIC Compiler

CONTENTS

Introduction	3-3
Creating Source Code	3-4
Compiling into Object Code	3-4
Linking a Program Together	3-5
Running Extended BASIC Programs	3-6
Memory Allocation in Extended BASIC	3-7
New File Types	3-7
The Extended BASIC Compiler	3-8
Installation Overview	3-8
Installation	3-9
Running the Extended BASIC Compiler Program	3-11
Using the Extended BASIC Compiler Program	3-12
Exiting the Extended BASIC Compiler Program	3-16
Extended BASIC Compiler Options	3-17
Extended Language Syntax: The /E Option	3-17
Integer Conversion: The /I Option	3-17
No Line Numbers: The /NL Option	3-18
No Markers: The /NM Option	3-18
Extended BASIC Compiler Errors	3-19
Linking The Object Files	3-22
Overview of the Linkage Process	3-22
Linking a Program	3-23
Using the Command File	3-25

INTRODUCTION

This section describes the creation of an Extended BASIC program and its conversion into machine language for execution by the Instrument Controller. In addition to your BASIC program, this process requires the following programs:

- Extended Basic Compiler (XBC.FD2)
- Linkage Utilities (XLL.FD2, XLM.FD2)
- Extended BASIC Runtime System (BSXRUN.FD2)

The programs listed are the minimum needed to compile and run an Extended BASIC program. Other programs may be needed for more complex programs.

NOTE

For Compiled BASIC users:

Extended BASIC programs are compiled in much the same way as Compiled BASIC programs. Utility programs, such as EDIT and FUP, are still used with Extended BASIC. The only changes in program utility tools involve programs that must generate, use or manipulate the new library formats and the Extended BASIC object format, OBX.

Creating Source Code

Extended BASIC language source code is created and modified using the System Editor program (EDIT.FD2). The original (source) program is written in the Extended BASIC language, using the program elements described in Section 2 of this manual and the Reference volume of the manual set. The System Editor (EDIT program), used to modify the program, is described in the 1722A System Guide.

Source programs may also be created and modified using the Edit mode in the BASIC Interpreter program (BASIC.FD2). Programs created or modified using the BASIC Interpreter program may only contain statements and syntax that are legal for Interpreted BASIC programs.

Compiling into Object Code

After the source program is created in the Extended BASIC language, it must be translated into object code. During the translation process, the compiler program performs some diagnostic functions and checks for Extended BASIC language syntax errors. If errors occur, the System Editor program is used to make corrections (while using the manual set for reference, as necessary). The Extended BASIC Compiler program has options that are entered in its command line to process optional language features. The Extended BASIC Compiler's command-line options are described later in this section.

The Extended BASIC Compiler will accept any of the following program forms as the source file:

- A main program (.BAS) alone.
- A main program followed by true subroutines.
- One or more true subroutines.

The Extended BASIC Compiler, XBC, is almost identical in operation to the standard BASIC Compiler, BC. The visible differences are:

- The “/A” (ASCII output file) option has no effect (but is still accepted).
- The output file extension has been changed from .OBJ to .OBX.

Linking a Program Together

After compiling, object files must be linked together to make an executable file using XLL, the Extended Linking Loader program. XLL combines program sections into a main program segment, combines subroutines with the main program, and adjusts memory references in the program. XLL replaces the LL and LE programs used with Compiled BASIC, FORTRAN, and Assembly. XLL has the capability to link FORTRAN and Assembly subroutines with Extended BASIC programs. The XLL program also provides a load map indicating the amount of extended memory used by XBASIC programs.

Extended BASIC programs are combined into libraries using the Extended Library Manager, XLM. XLM replaces the LM program used with Compiled BASIC, FORTRAN, and Assembly. It operates in much the same way as the LM program does. Refer to the Linkage Utilities section for more information.

Running Extended BASIC Programs

Extended BASIC programs will not run directly on the Instrument Controller. The output format of the Extended BASIC Compiler is an intermediate code rather than actual machine code. The Extended BASIC Runtime System program (BSXRUN.FD2) interprets this intermediate code at runtime. Although the Runtime System program interprets the intermediate code produced by the compiler, execution time is reduced because all of the syntax checking has been already done by the Extended BASIC Compiler program.

The Runtime System program checks for operational errors in the Extended BASIC programs as they run. Operational errors may either be reported to the user with error messages or processed internally, depending upon the program. Errors should be corrected using a text editor program, and the compile and link edit process should be repeated.

BSXRUN accepts the extended address pointers used in Extended BASIC, but otherwise appears identical to Compiled BASIC's BSCRUN program. Enough extended memory (NOT allocated as E-disk) must be available for XBASIC programs. The amount of extended memory required is given on the load map produced by the XLL program. Refer to Section 4 for a thorough explanation of the Runtime System.

Memory Allocation in Extended BASIC

Memory use affects the architecture of programs executed using Extended BASIC. There are two memory regions in Extended BASIC: the machine code segment and the BASIC code segment.

The machine code segment is a 64K byte memory region in which any machine language (Assembly or FORTRAN) modules are placed, together with any global variables. The machine code segment size is determined by the 16-bit address used by the TMS-99000 processor. This segment contains any machine language subroutines, BASIC variables, and the BSXRUN program. The size of this segment size limits the number of Assembly and FORTRAN subroutines that may be used and the amount of data (variables and main-memory arrays) that may be used at any one time.

The BASIC code segment is a much larger memory region into which the BASIC main program and subroutines are placed. The BASIC code segment size is limited by the amount of memory installed in the 1722A, which may not exceed 4 megabytes. Note that the (approximately) 32K bytes required for FDOS, the 64K bytes for the machine code segment, and any memory used for E-disk will reduce the amount of memory available for the BASIC code segment.

New File Types

Two new file formats, .OBX and .LIB, are designed for easy generation and quick linking, both important issues when you are generating large programs.

The use of extended memory has required the definition of a new object format. The binary format of Extended BASIC object files (.OBX) is easily manipulated by the XLL and XLM programs. Due to the flexibility provided by the .OBX file format, global variables are supported. .OBX files are a counterpart to the .OBJ files produced by other language processors.

.LBX files are Extended BASIC object library files. These files are created by the XLM library generator and are random libraries, unlike the .LIB library files in Compiled BASIC, which are sequential. For more information on XLM and XLL, see the Linkage Utilities section.

THE EXTENDED BASIC COMPILER

The Extended BASIC Compiler program is part of the Fluke 17XXA-205 Extended BASIC Language Option Packages for the Fluke 1722A series Instrument Controllers. The compiler program is one of several programs supplied with the language option.

The file name of the Extended BASIC Compiler program is XBC.FD2. The program is contained on a 5-1/4 inch floppy disk.

Installation Overview

For small programs, the Extended BASIC Compiler program and the source program to be compiled may be installed on any file-structured device prior to compilation. A high-capacity, file-structured device capable of high-speed I/O, such as the E-Disk, assigned as the System Device, will yield the fastest compilation time with a minimum of keyboard input. For large programs, the Extended BASIC Compiler program should remain on the floppy disk with the source program on the E-Disk, thereby reserving maximum memory space on the E-Disk for temporary and output files.

Installation

To install the Extended BASIC Compiler program and Extended BASIC source program in the suggested large-program configuration, follow these steps: (Be sure to terminate the command line by pressing the `<RETURN>` key.)

1. Set the instrument Power Switch to ON.
2. Insert a floppy disk that contains the current versions of the system software. The system software will complete the initialization process and load FDOS, resulting in the FDOS prompt.
3. Enter the File Utility program and configure the E-Disk from FDOS by typing:

```
FUP  
ED0: /Cnn
```

where nn is a large enough number of blocks to ensure adequate storage capacity for the source program and all scratch files.

The Extended BASIC Compiler requires 64K of user space for execution. The E-Disk must be configured so that 64K (32 blocks) of user space remains after configuration. If there is insufficient RAM available for an adequately sized E-Disk, then additional file-structured storage is required.

CAUTION

Configuring the E-Disk will destroy any files that may already be on the E-Disk. If files are on the E-Disk, do not configure the E-Disk before saving these files on a floppy disk or determining that they are expendable. See the 1722A System Guide if you need help in saving the files.

NOTE

If files are already on the E-Disk, it is already configured. Instead of configuring the E-Disk, type the File Utility command to zero the E-Disk.

```
ED0: /Z<RETURN>
```

4. Insert the disk containing the Extended BASIC source program into the disk drive.

5. Install the source program on the E-Disk by typing

`EDO: =<source program name>RETURN`

6. Assign the E-Disk as the system device (SY0:) by typing

`EDO: /ARETURN`

7. Insert a disk containing the BASIC Compiler program into the disk drive.

8. Verify that the files copied to the E-Disk were indeed copied by typing

`/Q`

9. Exit the File Utility program by typing

`/XRETURN`

NOTE

The Extended BASIC Compiler program consists of several overlaid program segments that share the same sections of memory. If the Extended BASIC Compiler program is being used from the floppy disk drive (MF0:), the disk must remain in the disk drive while the program is running to ensure that portions of the program are available as they are needed. Do NOT remove the disk from the drive while the program is running.

The software is now in the suggested configuration for maximum use of memory. Other configurations are also possible. See the System Guide if you need help in using the File Utility program to install other configurations.

With the software installed as desired, refer to the following paragraph, which describes how to enter and use the Extended BASIC Compiler program.

Running the Extended BASIC Compiler Program

If the Extended BASIC Compiler program is installed on the system device (SY0:), type the file name

XBC

to run the Extended BASIC Compiler program from FDOS.

If the instructions for installing the Extended BASIC Compiler program in the large-program configuration listed above are followed, the program will not be on the system device. If the program is on the floppy disk drive as suggested, type

MF0:XBC

In either case, do not type the file name extension .FD2.

As with other software for the Controller, the Extended BASIC Compiler program may be controlled by a command file. A command file is a series of operator inputs contained in a text file. The command file is initiated from FDOS, after which it controls program execution just as if the commands originated at the keyboard. A command file for implementing an Extended BASIC program, including inputs for this Extended BASIC Compiler program, is part of the software that makes up Extended BASIC. Command files may be terminated under certain error conditions (see BASIC Compiler Errors).

The Extended BASIC Compiler program will not work with the lexical code created by the RESAVEL or SAVEL command in Interpreted BASIC.

When you first enter the Extended BASIC Compiler program, it displays:

```
Extended BASIC Compiler  Version (x.y)
XBC>
```

NOTE

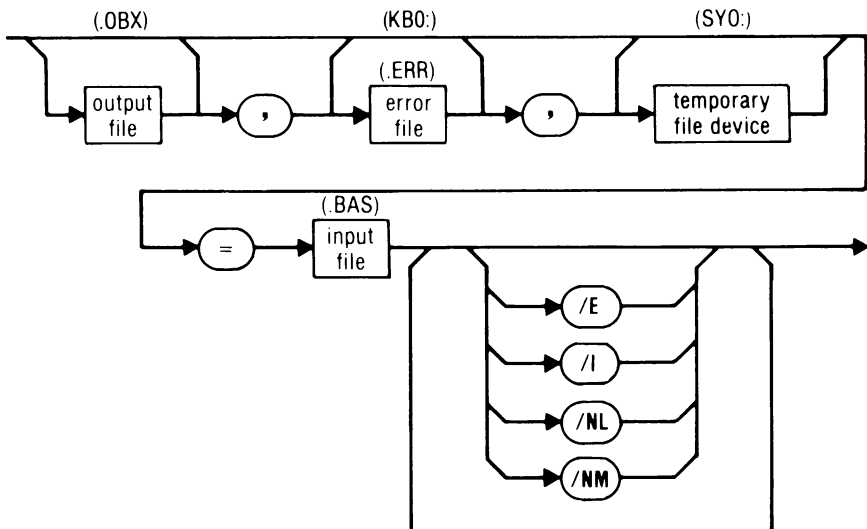
Verify that the version number (x.y) matches the number on the front of this manual. If it does not, call a Fluke Customer Service Center for advice.

Using the Extended BASIC Compiler Program

The Extended BASIC Compiler program translates a source program that is written in the Extended BASIC language into an object file. The source program is an ASCII file previously created with the Editor program included on the system disk. The source program may also be a BASIC program created by the BASIC Interpreter program.

If a subset of the BASIC language is used that is common to both the interpreted and compiled versions, the BASIC interpreter program may be used in the EDIT mode to create a source program. Note that the SAVE command must be used to store programs in an ASCII format.

In response to the Extended BASIC program prompt XBC>, type a command line specifying the names of the object file to be produced, the error file (if any), the temporary file device, the input file, and the options to be affected (see Extended BASIC Compiler Options), using the following syntax:



- ❑ The name of the output file is optional. If a file name is not specified, the file name will be created with the same name as the input file name and the extension .OBX.
- ❑ The output file name extension is optional. The default extension for the output file name is .OBX.
- ❑ The error file name is optional. If a file name is not specified, errors are printed on the display (KB0:). If an error file name is specified, its file name extension defaults to .ERR unless you specify otherwise.
- ❑ The temporary device name is optional. The default temporary device is the System Device (SY0:).
- ❑ The input file name is mandatory.
- ❑ The input file name extension is optional. The default extension for the input file is .BAS.
- ❑ Any number of options may be included, up to the maximum space available on the command line.
- ❑ Multiple options may be specified in any order.
- ❑ If multiple options are used, they are entered at the end of the command line without separation by commas or spaces. For example:

{command line}/E/NL/NM

- ❑ If a {?} is typed by itself in response to the Extended BASIC Compiler program prompt, this summary of the command syntax is displayed:

Command format: [Object][,[Error][,Tmp:]]=Source[/Switches]

Object	the output object (.OBX) file name (optional)
Error	the output error (.ERR) file name (optional)
Tmp:	the device to use for scratch files (optional)
=Source	the input source (.BAS) file name (required)

The Switches (all optional) are:

/E	permits the use of long variable names
/I	converts numeric constants to integer where possible
/NL	permits line numbers in the source program to be omitted
/NM	omits line number markers in the compiled output

The output file is the file that will contain the compiled program after the Extended BASIC Compiler program is finished. The error file will contain a list of any compilation errors that occurred during compilation. Temporary files will be created on the specified temporary device, then deleted before compilation is finished. The input file is the Extended BASIC source program that is to be compiled.

The optional names and default file name extensions allow a program to be compiled with minimal keyboard input. For example, the command line

```
, TEST=TEST
```

creates an output file named TEST.OBX and an error file named TEST.ERR. Note that no comma is necessary after the error file specification to denote the default temporary device. The command line

```
, , ED0: =TEST
```

specifies that an output file named TEST be created with the default file name extension .OBX. Errors will be printed to KB0:, the default. The E-Disk (ED0:) will be used for the temporary files. The command line

```
, RUN1=DEMO.SRC/E
```

creates an output file named DEMO.OBX, with the same file name as the input file but with the default output file name extension of .OBX. An error file is created with the specified name and the default error file name extension (RUN1.ERR). The output files are created from the specified DEMO.SRC input file. The /E option denotes that this input file is an extended format file. (See Compiler Options.)

When the command line is terminated with a `<RETURN>`, the Extended BASIC Compiler program begins compiling the source program. The Extended BASIC source code statements are analyzed and translated into an object format that is compatible with the Linking Loader program.

If errors are encountered during compilation, error messages are printed on the display. If an error file is named in the command line, the error messages are listed in that file (rather than on the display). Error messages are discussed later in this section.

Temporary files, which the Extended BASIC Compiler program creates during compilation, are removed from the disk before the compiler program finishes.

Exiting the Extended BASIC Compiler Program

The Extended BASIC Compiler program returns control to FDOS after compilation is completed. Type `<CTRL>/Z` to exit from the Extended BASIC Compiler program prompt. Type `<CTRL>/C` to exit the Extended BASIC Compiler program at any other time.

NOTE

If a `<CTRL>/P` is used to exit the Extended BASIC Compiler program, the temporary files that are used during compilation may be left on the temporary file device. If the Extended BASIC Compiler program is used again, these files will be automatically overwritten. However, some additional device space will be required. If this additional space is not available, or if the Extended BASIC Compiler program is not used again, the temporary file should be deleted. Use the File Utility program (FUP.FD2) to check for these files (\$BC00\$.TMP and \$BC01\$.TMP) and to delete them if they are present. Refer to the 1722A System Guide for help.

EXTENDED BASIC COMPILER OPTIONS

The Extended BASIC language contains several optional features designed to allow the programmer more flexibility in controlling input and output file format and in using memory space. The Extended BASIC Compiler program uses command-line option strings (switches) to accommodate each of these language options. See the syntax diagram under Using the Extended BASIC Compiler Program for use of the command-line options.

Extended Language Syntax: The /E Option

The /E option specifies that the source code contains extended language features such as continuation lines, statement labels, long variable identifiers (names), true subroutines, and extended flow of control statements. When the extended language syntax language option is used, keywords and numbers must be separated by spaces, tabs, or punctuation. This restriction on syntax makes the translating task easier and reduces compilation time.

The command file used with the Extended Syntax option is XBCE.CMD. This command file is similar to XBCC.CMD.

Integer Conversion: The /I Option

The /I option directs the compiler program to convert floating-point constants to integers where possible. Integer constants occupy less memory space and permit faster execution than do floating-point constants. Under control of the /I option, the floating-point value 5 will be treated internally as if it had been specified as an integer (5%). Whenever it is desired that a floating-point value be used as a value rather than as an integer, it should be written in a floating-point format exclusively. For example, write 5 as 5. or as 5E0 to protect it from being converted to an integer.

No Line Numbers: The /NL Option

The /NL option specifies that there are no line numbers in the input program. This option allows the programmer the convenience of not having to use line numbers for each line in a source program.

The /NL option must be specified if the source program does not have line numbers. If the /NL option is not specified, Missing Line Number error messages will result.

When line numbers are not used in a program, error messages that would normally refer to line numbers refer to sequential lines of text, beginning with line 1. For example, an error reported in line 4 does not refer to a line labeled 4, but to the fourth line from the top of the program.

No Markers: The /NM Option

The /NM option specifies that no line number markers are to be placed in the output file. Normally, the Extended BASIC Compiler program places markers in the output file to allow error messages to refer to specific line numbers. These markers occupy additional memory space that may be at a premium. The /NM option allows the user to create an output file that does not contain these markers. If the /NM option is used, error messages, interrupt messages, and stop messages will report line number 0.

EXTENDED BASIC COMPILER ERRORS

The Extended BASIC Compiler program detects errors in BASIC language syntax and data types during compilation. If the program finds errors it displays error messages on the screen as they occur. In addition, errors may be recorded in an error file. If you desire an error file, specify its name in the command line that configures compilation. See Entering the Extended BASIC Compiler Program for a description of the command line syntax required.

If you entered the Extended BASIC Compiler with a command file, and if an error is encountered during compilation, the command file will be terminated after completion of the Compiler program. This prevents the command file from attempting to link faulty object files and allows the programmer to correct the errors.

There are two types of errors that may be reported from the Extended BASIC Compiler program. Some of the BASIC language errors are of the form

Error number (xxx) at line (xxx)

Section 6 of this manual contains a description of these errors. Other errors that may be reported from the Extended BASIC Compiler program are listed in Table 3-1.

Table 3-1. Compiler Error Messages

(array name) must be parameter for DIM ()
Cycle length illegal for WBYTE
DIM () illegal for \$MAIN\$
Duplicate formal parameter (identifier)
Duplicate line number (line number)
Duplicate subroutine (subroutine name)
END in subroutine: use SUBEND instead

I/O error: ?Illegal file descriptor — Stop
I/O error: Channel in use — Stop
I/O error: Channel not open — Stop
I/O error: Device hardware error — Stop
I/O error: Device name mismatch — Stop
I/O error: Device not ready — Stop
I/O error: File does not exist — Stop
I/O error: File medium was swapped — Stop
I/O error: Ill-formed filename — Stop
I/O error: Medium write protected — Stop
I/O error: No channels available — Stop
I/O error: No room on device — Stop
I/O error: Non-existent memory — Stop
I/O error: RS-232 buffer overrun — Stop
I/O error: Read/write past physical end of file — Stop

Ill-formed numeric constant
Illegal LOCAL statement
Illegal LOCKOUT statement
Illegal RBIN statement
Illegal READ statement
Illegal REMOTE statement
Illegal RESTORE statement
Illegal RESUME statement
Illegal SIZE statement
Illegal STOP statement
Illegal SUB statement
Illegal TRIG statement
Illegal WBIN statement
Illegal channel
Illegal character '(character)' in line (line number)
Illegal time value

Table 3-1. Compiler Error Messages (cont)

Integer constant too large
Label (identifier) never defined
Label (identifier) redefined
Label for subroutine illegal
Line number (line number) out of order
MEMORY OVERFLOW (GEN), Stop
MEMORY OVERFLOW (PARSE), Stop
Mismatched parameter type for function (identifier)
Missing line number
Missing SUBEND statement
ON .. SUBRET illegal in \$MAIN\$
Statement(s) following SUBEND
SUBEND illegal in \$MAIN\$
SUBRET illegal in \$MAIN\$
Too many subroutines!

The following error message denotes an error that occurred in the Extended BASIC Compiler program itself. If such an error occurs, contact a Fluke Customer Service Center for help.

Compiler error — <internal routine name>, Stop

LINKING THE OBJECT FILES

After the Extended BASIC source program has been successfully compiled into an object file, it must be linked. The linkage process is also used to include separately generated program modules that have been generated by the Extended BASIC Compiler, the Assembler, or the FORTRAN compiler.

This discussion of the linkage process is limited to linking an Extended BASIC program to form an executable program. Other aspects of the use and operation of the various Linkage Utility programs are discussed and described in the Linkage Utilities section of this manual.

Overview of the Linkage Process

As described earlier, the Extended BASIC Compiler produces an object file of intermediate code. The Linkage Utility program must first combine the source program's object file with several other program modules, then adjust memory references in order to make an executable program.

A program may be loaded into the Controller's memory and executed if the following object file is linked via the linkage process:

File name	Purpose
MYPROG.OBX	The source program's object file.

When this file is linked, the resulting program will automatically run the Runtime System. If errors occur during the linkage process refer to the Extended Linkage Utilities (Extended Linking Loader) section for help.

Linking a Program

The following paragraphs describe a generalized procedure for linking program modules together to create an executable program. Specific information for each of the Linkage Utility programs may be found in the Linkage Utilities section of this manual.

You will need the following software:

<programname>.OBX The Extended BASIC program
XLL.FD2 The Extended Linking Loader program

The Extended Linking Loader program must be used to link Extended BASIC programs, using the following procedure:

In the procedure, the command-line examples assume that the files needed are present on the System Device (SY0:). If this is not true, assign the appropriate device as the System Device or explicitly give the device designation with the file name. For example:

MF0:MYPROG.OBX

1. Run the Extended Linking Loader program from FDOS by typing:

XLL

2. Use the Include command to specify the program segments to be combined. Enter the following items at the XLL prompt:

I <programname>

For example, if the program to be compiled were named CLEVER.OBX, the XLL command-line would look like this:

I CLEVER

3. Use the O command to specify the file name for the executable file to be produced.

O <outputfilename>

4. Use the M command to specify the file name to be used for the MAP file. This file contains the module location map, table of external references, and any error messages. This step may be ignored for relatively simple programs that do not involve multiple object files.

M <mapfilename>

5. Use the G command to start the linkage process.

G

6. The Extended Linking Loader program will now begin the linkage process. When the program has finished, control will be returned to FDOS and the output of the Extended Linking Loader program will be in the file specified in step 3.
7. Run the completed program from FDOS by typing: ,

<outputfilename>

Note that the file name extension is not used.

Note also that linkage processes differ between Compiled BASIC and Extended BASIC programs. Extended BASIC programs do not use the routine B\$LOAD. The XLL program automatically puts an equivalent program module, {filename}.FD2, into the output file.

The compilation/linkage process is a natural use for a command file. A simple command file that helps to simplify and automate this function is described next.

USING THE COMMAND FILE

XBCC.CMD is a command file that creates simple (non-overlay structured) Extended BASIC programs. The command file contains the user inputs at each step for compiling and linking programs. Refer to the System Guide for an explanation of command files.

To begin the command file for creating a program, type

```
XBCC <filename>           !file name extension not required
```

in response to the FDOS> prompt. The command file will be used as a series of keyboard-like inputs, with the specified file name used wherever question marks appear.

The command file for creating Extended BASIC programs without overlays and without extended syntax is:

```
XBC          !load Extended BASIC Compiler
&Compiling... !display message
=?          !substitute file name and compile
XLL         !load linking loader
&Linking...  !display message
I ?         !substitute file name and link with runtime
            !system loader program
O ?         !XLL output to file name
M           !map to display
G           !go do it
FUP ?.OBX/D !remove object file
```

1. The first step in the command file loads and runs the Extended BASIC Compiler program.
2. Line 3 specifies the files to be created. The output file will have the same name as the input file, with the default extension .FD2. The error file will be printed on KB0:. Temporary files will be placed on the system device SY0:. The question mark (?) indicates that the name that was specified in the command line will be used here. The default file name extension (.BAS) is used for the source file.
3. Line 4 loads and runs the Extended Linking Loader program after the Extended BASIC Compiler program is finished.
4. Following that, the input to the Linking Loader is specified as the output file from the Extended BASIC Compiler program ((name).OBX).

5. The output of the Extended Linking Loader is specified as the original file name with the default extension .FD2 in line 8.
6. Line 9 directs the map file to be printed on the screen.
7. Line 10 begins processing of the Extended Linking Loader program.
8. After the Extended Linking Loader program finishes and returns control to FDOS, the next line loads and runs the File Utility program.
9. The File Utility program is used to delete the .OBX file that was created by the compiler program and used for input by the Extended Linking Loader program. The object file is no longer needed.
10. Control returns to FDOS.

Section 4

Extended BASIC Runtime System

CONTENTS

Introduction	4-3
Running the Extended BASIC Runtime System Program	4-3
Running the Runtime System Program Automatically	4-3
Running the Runtime System Program from the Keyboard	4-4
Using the Runtime System Program	4-6
Exiting the Extended BASIC Runtime System	4-7
Runtime System Messages	4-8
Runtime System Error Checking	4-9

INTRODUCTION

The Extended BASIC Runtime System program provides a runtime environment to run BASIC programs in image file form. The Runtime System program is needed because Extended BASIC programs are not machine language programs, and the 17XXA Series Instrument Controllers cannot execute them directly. The Runtime System program runs at the same time as Extended BASIC programs (hence the term Runtime System) and adapts the Controller so that it will run them.

RUNNING THE EXTENDED BASIC RUNTIME SYSTEM PROGRAM

The Extended BASIC Runtime System program may be run automatically by running a correctly compiled and linked Extended BASIC program or it may be run independently from FDOS like other programs.

Running the Runtime System Program Automatically

A standard sub-program must be linked to an Extended BASIC program that automatically loads and executes the Runtime System program. The standard sub-program is included by the Extended Linking Loader program.

The Runtime System loader program automatically searches through all of the file-structured devices for the Runtime System program. If the Runtime System program is found, it is loaded and run automatically, after which the Extended BASIC program is run. If the Runtime System is not found, the message

!No BASIC Runtime System

is printed, and control returns to FDOS.

If the message

?System Error

is printed, some kind of error has happened while you are trying to load the Runtime System. Trying to run the Runtime System program from the FDOS> prompt may be the reason the Runtime System program couldn't be loaded.

Running the Runtime System Program from the Keyboard

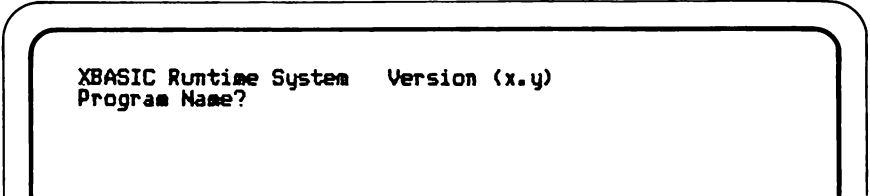
There are two ways that you can run the Extended BASIC Runtime System program from the keyboard:

- By entering the program name by itself
- By combining the name of the Runtime System program with the name of the Extended BASIC program

The first method is the same as in other Instrument Controller programs. Type the program name in response to the FDOS prompt:

BSXRUN

The Runtime System will respond with

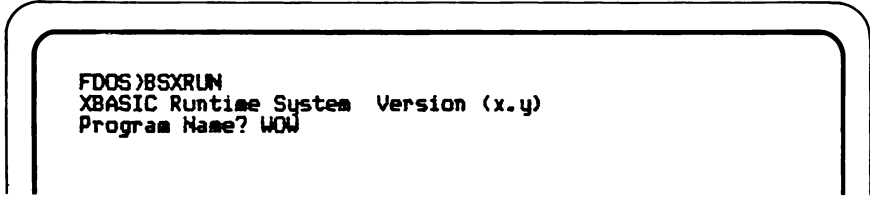


```
XBASIC Runtime System  Version (x.y)
Program Name?
```

NOTE

Verify that the version number (x.y) matches the version number on the front of this manual. If the version number does not match, contact a Fluke Customer Service Center for advice.

For example, to run a program named WOW.FD2, type



```
FDOS>BSXRUN
XBASIC Runtime System  Version (x.y)
Program Name? WOW
```

If the Runtime System program is not available, FDOS will print

?File not found

The alternate way to enter the Runtime System program is to include the name of the Runtime System program and the Extended BASIC program together on the same FDOS command line. For example,

FDOS>BSXRUN WDW

If the Runtime System program is not available, FDOS will print the error message above.

USING THE RUNTIME SYSTEM PROGRAM

If the Runtime System is entered without a program name on the command line, the program will prompt the user for the name of a compiled BASIC program to run. Enter the name of the Extended BASIC program in response to this prompt.

If the alternate entry method is used, combining the names of the Runtime System and program in a single command line, the Runtime System will already have the program name.

The Runtime System program will search for this program and execute it. If the program cannot be found, the Runtime System program will print

?Program not found

followed by a repeat of the prompt

Program name?

Once the Extended BASIC program is found, no further interaction occurs with the user, except for error processing as described later in this section.

EXITING THE EXTENDED BASIC RUNTIME SYSTEM

The Extended BASIC Runtime System program returns control to FDOS whenever the Extended BASIC program being run executes an END statement, an EXIT statement, or a STOP statement.

To exit the Extended BASIC Runtime System program from a "Program name?" prompt, type

`<CTRL>/Z` or `<CTRL>/P`

Control also returns to FDOS after a Recoverable error occurs if an error handling routine is not used.

If the BASIC program does not include a CTRL/C handler, you can enter `<CTRL>/C` from the keyboard to return control to FDOS. If the console is in ECHO mode, `<CTRL>/P` will also return control to FDOS. If a CTRL/C handler does exist, and the console is in NOECHO mode, you can enter `<CTRL>/P` from the keyboard to branch to the CTRL/C handler.

RUNTIME SYSTEM MESSAGES

The Runtime System program prints messages on the screen to inform the user whenever a STOP statement occurs, the ABORT key is pressed, a <CTRL>/C is entered (unless an ON CTRL/C-interrupt handling statement is used), or when errors occur.

Whenever a STOP statement occurs in a program, this message is printed on the display

STOP at line <line number> in module <module name>

and control returns to FDOS.

An untrapped <CTRL>/C input or an ABORT key causes the Runtime System program to print

!Abort at line <line number> in module <module name>

on the display and return control to FDOS.

The messages that are printed in response to errors are described below under Runtime System Error Checking.

NOTE

If the No Markers option is used for compiling the program, all system messages will refer to line zero.

If the No Line number optional syntax is used, line numbers will refer to program lines in sequence. For example, line 14 will refer to the fourteenth line from the top of the program.

RUNTIME SYSTEM ERROR CHECKING

The Extended BASIC Runtime System program monitors the Extended BASIC program for language errors that occur while the program is running (syntax errors have already been diagnosed by the Extended BASIC Compiler program). Fatal errors will cause the Runtime System to print an error message and terminate the program, returning control to FDOS. The message for a fatal error is

!Error <error number> at line <line number> in module <module name>

Recoverable errors will either cause an error message to be printed, or if an ON ERROR interrupt processing statement is pending, will cause control to be transferred to the error processing routine. A recoverable error will have the format

?Error <error number> at line <line number> in module <module name>

If an error handling routine is used, no error message is printed.

NOTE

If the No Markers option is used for compiling the program, all system messages will refer to line zero.

If the No Line number optional syntax is used, line numbers will refer to program lines in sequence. For example, line 14 will refer to the fourteenth line from the top of the program.

Section 6 of this manual contains a list of the error numbers that may be reported by the Runtime System program.

Section 5

Extended BASIC Linkage Utilities

CONTENTS

Introduction	5-3
The Extended BASIC Linking Loader Program	5-4
Executing the XLL Program	5-4
Terminating the XLL Program	5-4
Using the XLL Program	5-5
XLL File Name Conventions	5-7
Extended Linking Loader Commands	5-8
END/GO Command	5-10
FIND Command	5-11
INCLUDE Command	5-13
MAP Command	5-15
OUTPUT Command	5-17
Extended BASIC Linking Loader Error Messages	5-18
Extended BASIC Linking Loader Map Format	5-19
Module List	5-20
Symbol Table	5-22
Common Symbol Table	5-24
Error Message Table	5-24
The Extended BASIC Library Manager Program	5-25
Executing the XLM Program	5-25
Terminating the XLM Program	5-25
Using the XLM Program	5-26
XLM File Name Conventions	5-27

CONTENTS, *continued*

Extended BASIC Library Manager Commands 5-28

 /C - Copy 5-30

 /D - Delete 5-31

 /E - Extended List 5-32

 /L - List 5-34

 /M - Merge 5-35

 -X - Exit 5-36

 ? - Help 5-37

Extended BASIC Library Manager Error Messages 5-38

INTRODUCTION

Two linkage utility programs are available with Extended BASIC on the Fluke 1722A Instrument Controller:

- The Extended Linking Loader Program, XLL
- The Extended Library Manager Program, XLM

The Extended Linking Loader program is used to link Extended BASIC programs. XLL combines program sections into a main program segment, combines subroutines with the main program, and adjusts memory references in the program. XLL replaces the LL and LE programs used with the Compiled BASIC, FORTRAN, and Assembly languages.

The Extended Library Manager is used to combine Extended BASIC programs into libraries. XLM replaces the LM program used with the Compiled BASIC, FORTRAN, and Assembly languages. XLM operates in much the same way as the LM program does, but it creates random, rather than sequential libraries.

THE EXTENDED BASIC LINKING LOADER PROGRAM

The Extended BASIC Linking Loader program is a linkage editor that combines object modules produced by the Extended BASIC Compiler (XBC) into a single executable image file. XLL can also combine XBC object modules with object modules produced by the Assembler or the FORTRAN Compiler. Any of these object modules can be taken from libraries of previously compiled or assembled source programs.

The file name of the Extended BASIC Linking Loader program is XLL.FD2. This program is distributed on a 5-1/4 inch floppy disk with the rest of the Extended BASIC system.

CAUTION

The XLL program uses overlays. If XLL is loaded from a floppy disk, this disk MUST NOT be removed from the disk drive while XLL is running.

Executing the XLL Program

To execute the XLL program, from the FDOS command line prompt, type:

XLL<RETURN>

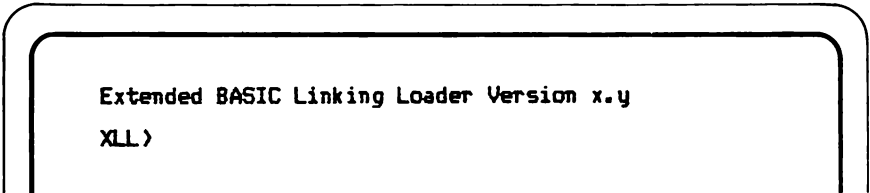
Terminating the XLL Program

The XLL program will automatically exit to FDOS when its processing is completed. To stop XLL at any time, type <CTRL>/P. To exit to FDOS, type <CTRL>/Z in response to the XLL prompt.

Using the XLL Program

The XLL program must be used to create an executable program (.FD2 file) from the .OBX (Extended BASIC object) files produced by the Extended BASIC Compiler program, XBC.

When first started the XLL program displays:



```
Extended BASIC Linking Loader Version x.y  
XLL >
```

NOTE

If the major version number (x) does not match the major version number on the front of this manual, call a Fluke Customer Service Center for advice.

Respond to XLL's prompts using Extended Linking Loader commands. These commands are detailed below.

1. Use the INCLUDE command to enter the names of the Extended BASIC object modules (.OBX files) and machine language object modules (.OBJ files) to be included in the final executable program.
2. Use the FIND command to enter the names of any Extended BASIC libraries (.LBX files, created by XLM) or machine language libraries (.LIB files, created by the Library Manager program, LM) which contain additional needed object modules.
3. Use the OUTPUT command to name the final executable image (.FD2) file.
4. Use the MAP command to request a load map if desired.
5. Use the GO or END command to start the linkage process.

Once the GO or END command has been given, XLL will proceed to link the modules together. All of the input files named by INCLUDE and FIND commands are searched once. This search is performed in the order in which the files were named by the INCLUDE and FIND commands. The command sequence

```
XLL>include file1.obx,file2.obx  
XLL>find lib1.lbx  
XLL>include file3.obj  
XLL>output image  
XLL>go
```

would cause XLL to search file1.obx, file2.obx, lib1.lbx, and file3.obj in exactly that order. When a library file is encountered in the input list, that library is searched once at that point. Any library modules that contain a definition for an unresolved external reference are scheduled for loading.

Once all of the input files and libraries have been examined, XLL assigns addresses to all modules, external symbols, and common regions and prints a load map (if requested). XLL will terminate execution at this point if any external symbols are undefined or multiply defined. XLL then rereads the input modules that will appear in the output file, creates the executable image (.FD2) file, and returns control to FDOS.

As soon as the FDOS> prompt is displayed, the file named by the OUTPUT command may be executed by typing its name.

XLL File Name Conventions

XLL permits the use of “wild card” characters to match input (INCLUDE and MAP command) file names. These wild cards are the asterisk “*”, which matches zero or more arbitrary characters in a file name or in an extension, and the question mark “?”, which matches any one character in a file name or extension. File names that contain wild card characters are called patterns.

Examples of file name patterns and the file names that they can match are listed below. Arbitrary characters are indicated with an “x”.

Pattern	Could Match
A*A	AA, AxA, AxxA, AxxxA, AxxxxA
A?A	AxA
A?*A	AxA, AxxA, AxxxA, AxxxxA
A?A*	AxA, AxAx, AxAxx, AxAxxx
A*B*A	ABA, AxBA, AxxBA, AxxxBA, ABxA, ABxxA, ABxxxA AxBxA, AxxBxA, AxBxxA, ...
A?B*A	AxBA, AxBxA, AxBxxA

There are two limitations on the use of wild cards with XLL.

- ☐ Wild cards may not be used in device names.
- ☐ Wild cards may not be used in output file names.

Extended Linking Loader Commands

XLL commands specify the names of files to be used during the linking process and are entered in response to the XLL> prompt. These commands are:

Command	Description
<CTRL>/Z	Terminate XLL and return to the FDOS> prompt.
?	Print a short description of the commands available.
END	End of specifications; begin linking.
FIND	Search these libraries for needed object modules.
GO	End of specifications; begin linking.
INCLUDE	Include object modules in these files.
MAP	Load map file specification.
OUTPUT	Output file specification.

Guidelines for these commands are:

- ❑ Only one command may be given on each line.
- ❑ Commands may be abbreviated. For example, the FIND command may be abbreviated as “F”, “FI”, or “FIN”.
- ❑ Commands and file names may be entered in upper case, lower case, or a mixture of both.
- ❑ At least one space must separate a command and any file names which follow it.
- ❑ At least one INCLUDE command must be given.
- ❑ The order of INCLUDE and FIND commands determines the sequence in which modules will be linked and libraries searched.

- ❑ The order and placement of the **OUTPUT** and **MAP** commands is not important, but at least one **OUTPUT** command must be given prior to the **END** or **GO** command.
- ❑ An **END** or **GO** command must be used to terminate command input and start the linking process.
- ❑ The syntax for file specifications (where required) is:

[device:] filename [.extension]

The device field is optional and defaults to one of two devices. Default will be to the last device name given on the command line. If no device name has been given on the command line, default will be to the system device, **SYO:**.

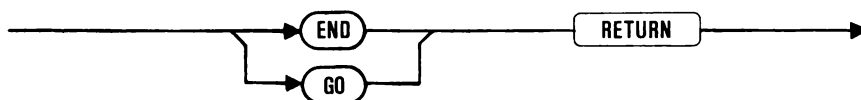
The extension is also optional. The default extensions applicable to any given command are detailed in the following sections.

END

GO

Terminate Command Input and Begin Linking

Syntax Diagram



Description

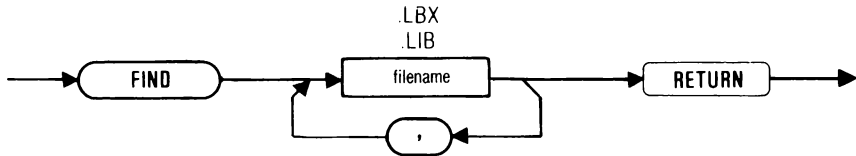
The END or GO command tells XLL that all of the necessary files and options have been specified and that the actual linking process should begin. If no INCLUDE or OUTPUT command has been given, XLL will say so and reissue the XLL> prompt.

XLL will not create an output file if any errors occur during the linkage process. If no errors occur, the output image will be written to the file named in the last OUTPUT command. Whether or not errors occur, XLL will return control to FDOS when its processing has finished.

FIND

Search Libraries for Needed Modules

Syntax Diagram



Description

FIND names library files that may contain modules to be linked with other files named in INCLUDE commands. Libraries are collections of standard and proven modules in object file format. (They have been compiled or assembled.) Libraries are created and modified through the use of library manager programs. See the CBASIC, Assembly, or FORTRAN manual, or the Extended BASIC Library Manager (Linkage Utilities section) in this manual for information on creating and modifying libraries.

- The FIND command is optional.
- More than one FIND command may be used.
- The FIND command name may be abbreviated to one or more letters.
- More than one library may be specified in a FIND command; multiple library file names are separated with commas.
- Library modules are linked only if they are needed. A library module is “needed” if the module contains a definition for a global symbol (for example, a subroutine or data item name) that has been referenced by a preceding module and that has not yet been defined by another module.

- The default extensions for library file names are .LIB (machine language library) and .LBX (Extended BASIC library). The command

XLL>find lib1

will match either or both of the library names “lib1.lbx” and “lib1.lib”. The library “lib1.lbx” will be searched first.

- Wild card file name patterns may be used to specify FIND file names.
- If one of the files named in the FIND command cannot be located, the linking loader will print

File not found

and the entire command line will be ignored.

All files named by the FIND command must reside on file-structured devices.

Examples

Two examples of FIND command usage follow.

XLL>find wd1:flib

Search a library “wd1:flib.lbx” (if present) and then a library named “wd1:flib.lib” (if present). If neither of these files exist, an error message will be printed.

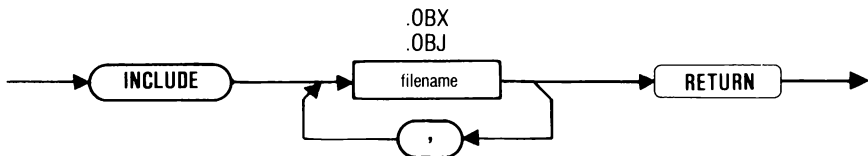
XLL>f mf0:gen*.lbx,src1lib.lib

Search all .LBX libraries on device MF0: whose names start with the characters “gen”, and then search the library “mf0:src1lib.lib”. Note that a device name specification in an input list applies to all file name patterns that follow until another device’s name appears.

INCLUDE

Unconditionally Include an Object File

Syntax Diagram



Description

The INCLUDE command specifies the names of object files that contain program segments to be combined. These object files may be machine language object (.OBJ) files created by the FORTRAN Compiler or the Assembler. They may also be Extended BASIC object (.OBX) files created by the Extended BASIC Compiler, XBC.

- ❑ The INCLUDE command name may be abbreviated to one or more letters.
- ❑ More than one file may be specified by each INCLUDE command; multiple file names are separated by commas.
- ❑ At least one INCLUDE command must be given.
- ❑ If multiple INCLUDE files are specified, they are linked in the order specified.
- ❑ Unlike library files (specified by the FIND command), the modules contained in INCLUDE files are linked whether or not they are referenced by other object modules.
- ❑ Exactly one of the INCLUDE files or FIND modules must contain a BASIC main program (a symbol named "\$MAIN\$").
- ❑ The default extensions supplied are .OBX for Extended BASIC object files and .OBJ for machine language object files.
- ❑ Machine language object (.OBJ) files may be in either ASCII or binary (compressed) format.

- If a machine language object library (.LIB) file is named in an INCLUDE command, all of the modules in that library will be loaded.
- An Extended BASIC object library (.LBX) file should not be used with an INCLUDE command. These libraries use a random organization that is not compatible with the .OBX file format and will generate a diagnostic when processed by XLL.
- Wild card file name patterns may be used to specify INCLUDE file names.
- If one of the files named in the INCLUDE command cannot be located, the linking loader will print

File not found

and the entire command line will be ignored.

- All files named by the INCLUDE command must reside on file-structured devices.

Examples

Two examples of INCLUDE command usage follow.

```
XLL>include tx*,ed0:mx?.obj
```

Include all .OBX and .OBJ files whose names start with the characters "tx" on device SY0:, as well as all .OBJ files whose names are three characters long and which start with the characters "mx" on device ED0:.

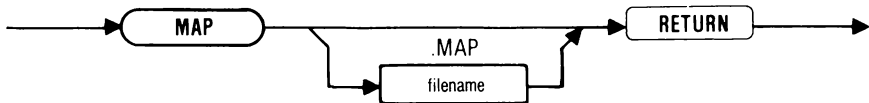
```
XLL>i mf0: lib1. lib, file2, wd0: file3. obx
```

Include the files "mf0:lib1.lib", "mf0:file2.obj" and/or "mf0:file2.obx", and "wd0:file3.obx". Note that a device name specification applies to all following file names until another device name is given.

MAP

Generate a Load Map

Syntax Diagram



Description

The MAP command requests that a load map of the generated program be created by XLL. This file contains:

1. A list of all modules included as part of the output image (.FD2) file, including the load address of each module.
2. An alphabetic list of all global symbols and their locations.
3. A list of all common variables and arrays, both blank and labeled, defined by the machine language object programs.
4. A list of all undefined or multiply-defined global symbols.

A list of rules for MAP command usage follows. See Extended BASIC Linking Loader Map Format, below, for a description of the map generated by XLL.

- The MAP command is optional.
- A linkage map will not be generated unless a MAP command is given.
- The MAP command may be abbreviated to one or more letters.
- Only one file name may be given with the MAP command.
- If no file name is given with the MAP command the map will be printed to the console (device KB0:).
- The default extension for the map file is .MAP.
- No wild card characters may be used in the map file specification.

Examples

Two examples of MAP command usage follow.

```
XLL>map
```

Print a load map to the console, device KB0:.

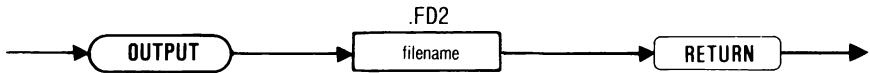
```
XLL>mb0:zippo
```

Print a load map to file “mb0:zippo.map”.

OUTPUT

Specify the Name of the Output File

Syntax Diagram



Description

The OUTPUT command specifies the name of the executable file to be produced. The output file is an image of an executable program which may be loaded and executed by the Extended BASIC Runtime System, BSXRUN.

- The OUTPUT command may be abbreviated to one or more letters.
- Only one output file name may be given with an OUTPUT command.
- At least one OUTPUT command must be given. If more than one is given the output file specified by the last OUTPUT command will be used.
- The default extension of the output file is .FD2.
- No wild card characters may be used in the output file specification.

Example

```
XLL>out test1
```

Write the output of XLL to file "sy0:test1.fd2".

Extended BASIC Linking Loader Error Messages

The XLL program operates in two phases: the input phase and the execution phase. The input phase takes place from the time XLL begins execution until an END or GO command is issued. The execution phase occurs between the time the END or GO command is issued and XLL returns to FDOS.

Errors that occur in the input phase are recoverable. The line for which the diagnostics are produced is ignored and may be re-entered with the necessary corrections.

NOTE

If the commands to XLL are taken from a command file, an input phase error will return control to FDOS and further command file input will be suppressed.

Errors that occur during execution are usually fatal. XLL will not create an output file if any of these errors occur. See the Error Messages section for lists of XLL general execution and I/O error messages.

Extended BASIC Linking Loader Map Format

The XLL program, under control of the MAP command, produces a detailed listing of the modules that have been linked together to produce an executable program. This listing is called a load map, and is stored in a print image file. The load map is in the following four parts:

1. The module list, which gives the names of the linked modules, along with their type, address, length, and other information.
2. The symbol table, which is an alphabetical list of all global symbols in the modules on the module list.
3. The common symbol table, which lists all common regions allocated.
4. A listing of error messages for all multi-defined or undefined symbols referenced by the program.

The map format is illustrated by an actual load map produced by XLL. The following commands were given to XLL, with WD1: used as the system device:

```
XLL>i ed0:main,wd1:alpha,beta,graph
XLL>f flib,util,bench
XLL>o ed0:main
XLL>m ed0:main
XLL>g
```

Module List

The first part of the load map, the module list, lists the names of the linked modules, their number, type, starting address, number of bytes used, date and time of compilation or assembly, name of the program that created the object module, and the file from which the module was read. Refer to Figure 5-1 for the example of the module list produced by XLL.

An explanation of each field follows:

Module	is the name of the module (module identifier, or IDT). For Extended BASIC programs this is the name of the .OBX file. Certain names, however, have special meanings. "\$GLOBAL" refers to memory reserved for EXPORT variables; "\$DATA" refers either to DATA statement memory or to data segment memory in Assembly or FORTRAN programs.
Number	is the ordinal number of the module in the input. Other parts of the load map use this number to identify the module in which a symbol was defined or referenced.
Type	is the module type. Currently, the types are: OBX for INCLUDE Extended BASIC modules; OBJ for INCLUDE Assembly or FORTRAN modules; LIB for FIND Assembly or FORTRAN modules; LBX for FIND Extended BASIC modules.
Base	is the starting address of the module in memory. Notice that 4-digit hexadecimal addresses refer to the 64K-byte machine code memory segment, and 8-digit hexadecimal addresses refer to the Extended BASIC memory segment. While each segment starts at address zero, only the machine-code segment will start at physical address zero when the program is executed; the Extended BASIC memory segment will actually be loaded into a different set of addresses by the BSXRUN program.
Length	is the number of bytes (in hexadecimal) used by a module. The length is a 4-digit number for modules loaded into the machine code segment, and is 8 digits for modules loaded into the Extended BASIC segment.

Date, Time are the date and time at which the module was compiled or assembled.

Creator is the name of the program that created the object module. Some common names are: LE for the Linkage Editor, ASM for the Assembler, FC for the FORTRAN Compiler, and XBC for the Extended BASIC Compiler.

File is the name of the file from which the module was read. This name is printed only for the first module taken from any particular file.

At the bottom of the page is a summary of the memory requirements for the complete program. Of particular interest is the "Extended memory required", which indicates how much free memory (which must not be allocated to E-disk) is required in order to execute the program.

Extended BASIC Linking Loader (XLL) Version 1.0 05-Oct-84 14:17 Page 1									
Module	Number	Type	Base	Length	Date	Time	Creator	File	
MAIN	1	OBJ	00000000	000000C6	05-Oct-84	14:16	XBC	ED0:MAIN.OBX	
\$GLOBAL			0000	0208					
ALPHA	2	OBJ	0208	003E	07-Aug-84	14:46	FC	WD1:ALPHA.OBJ	
\$DATA			0246	0034					
BETA	3	OBJ	027A	0036	07-Aug-84	14:47	FC	WD1:BETA.OBJ	
\$DATA			02B0	0034					
graph	4	OBJ	02E4	0230	05-Jan-84	14:30	LE	WD1:GRAPH.OBJ	
\$ritp	5	LIB	0514	0140	14-Sep-83	16:34	ASM	WD1:FLIB.LIB	
SIN	6	LIB	0654	01AE	14-Sep-81	13:47	ASM		
\$RWSP	7	LIB	0802	0000	14-Sep-83	16:18	ASM		
\$DATA			0802	00FA					
RELSV	8	LIB	08FC	0486	25-Aug-83	08:41	ASM		
\$EATL	9	LIB	0082	00C3	14-Sep-81	11:49	ASM		
\$ERRC	10	LIB	0E46	010C	15-Sep-81	08:24	ASM		
\$RPAU	11	LIB	0F52	008A	17-Jan-84	17:25	ASM		
\$XSUP	12	LIB	100C	00A6	09-Sep-83	13:09	ASM		
\$XLOG	13	LIB	1082	0052	19-Jan-83	10:24	ASM		
\$XIO	14	LIB	1104	040E	09-Sep-83	13:10	ASM		
\$RWURK	15	LIB	1512	0000	19-Jan-83	10:21	ASM		
\$DATA			1512	0100					
\$XFCB	16	LIB	1612	0000	19-Jan-83	10:21	ASM		
\$DATA			1612	01C2					
STARR	17	LBX	000000C6	00006292	05-Oct-84	14:01	XBC	WD1:UTIL.LBX	
TEXT17	18	LBX	00006358	00004DF6	04-Oct-84	08:06	XBC		
\$DATA			0000814E	00000075					
BLKJCK	19	LBX	000081C4	000025AA	05-Oct-84	14:01	XBC		
GEP	20	LBX	0000076E	00004C2A	04-Oct-84	13:18	XBC		
BUBBLE	21	LBX	00012398	000001EA	05-Oct-84	14:13	XBC	WD1:BENCH.LBX	
TICTOC	22	LBX	00012582	0000027A	05-Oct-84	09:09	XBC		
SIEVE	23	LBX	000127FC	000001A4	05-Oct-84	14:12	XBC		
Machine code segment size: 6160 bytes (6.0 Kbytes)									
BASIC code segment size: 76192 bytes (74.4 Kbytes)									
Extended memory required: 152 blocks (76 Kbytes)									

Figure 5-1. Module List

Symbol Table

The symbol table is an alphabetical list of all global symbols in the linked modules. The first field contains the name of the global identifier. This field is ordered alphabetically, except when linking very large programs. Other fields indicate the symbol type, hexadecimal value, and module number in which the symbol is defined. Refer to Figure 5-2 for an example of the Symbol Table Format.

An explanation of each field follows:

- | | |
|--------|--|
| Name | is the global identifier itself. For EXPORTed (global) BASIC variables, these names resemble the forms in the source program: a floating-point variable is marked with a trailing "#"; arrays are marked with trailing brackets "[]". |
| Type | <p>is a one-letter code that indicates the symbol type.
These codes are:</p> <ul style="list-style-type: none">"B" for a BASIC subroutine name,"C" for a labeled COMMON segment name,"G" for a BASIC global variable,"O" for an Assembly or FORTRAN subroutine or global data item name,"U" for an undefined symbol. <p>A letter code may be followed by an asterisk "*", which indicates that the symbol is not referenced by another module.</p> |
| Value | is the value assigned to a symbol, expressed as a hexadecimal number. For BASIC subroutines this is an 8-digit (extended memory) address. An absolute value is printed with a trailing asterisk, "*". |
| Number | is the module number in which the symbol is defined. |

Extended BASIC Linking Loader (XLL) Version 1.0 05-Oct-84 14:17 Page 2

Name	Type	Value	Number	Name	Type	Value	Number
\$MAIN\$	B	0000002A	1	ABC	C	17D4	
ALPHA	O	0208	2	BETA	O	027A	3
BLKJCK	B	00008604	19	BUBBLE	B	00012422	21
COS	OM	0654	6	COUNTZ	GM	0004	1
COUNTERSZ[]	GM	003E	1	DEVICES[]	GM	000E	1
DOT	O	02E4	4	DRAW	O	0312	4
ERAGRP	O	034C	4	F\$BUFS	O	17D2	16
F\$EATL	O	0D82	9	F\$EBAZ	OM	0D88	9
F\$EDBZ	O	0DA6	9	F\$EINA	O	0DAC	9
F\$ERRC	O	0E46	10	F\$ERRS	O	0E4C	10
F\$ERRT	O	0F7C	11	F\$ESCL	OM	0D82	9
F\$IRLD	O	0956	8	F\$IRMC	O	0918	8
F\$IRML	O	091A	8	F\$IRNV	OM	0976	8
F\$IRNM	OM	0A56	8	F\$IRPK	O	0A72	8
F\$IRSB	O	090C	8	F\$IRSC	O	090A	8
F\$IRSS	O	0C5C	8	F\$IRST	O	0966	8
F\$RDBL	OM	0652	5	F\$RERB	O	1544	15
F\$RGMY	O	04D2	4	F\$RITP	O	0514	5
F\$RPAU	OM	0F52	11	F\$RREL	O	0652	5
F\$RURK	O	1512	15	F\$RWSP	O	0802	7
F\$XBC	O	00FE	15	F\$XCLF	OM	1014	12
F\$XCLS	O	100C	12	F\$XCOL	OM	00FA	15
F\$XEDF	O	134C	14	F\$XERR	O	1084	12
F\$XFCB	O	1612	16	F\$XFCE	O	17CA	16
F\$XFLN	OM	0016	16	F\$XFLS	O	14A2	14
F\$XLDG	O	1082	13	F\$XPSE	O	1070	12
F\$XRED	OM	1104	14	F\$XRIR	O	0000	14
F\$XRND	OM	153A	15	F\$XRST	O	108E	12
F\$XSLI	O	0000	14	F\$XSTI	O	0000	14
F\$XSTP	O	107C	12	F\$XTRA	OM	1088	12
F\$XHIR	O	0000	14	F\$XWRT	OM	1200	14
F\$XUSO	OM	0828	7	FILENAME\$	GM	0000	1
GEP	B	0000E444	20	GRPOFF	O	036E	4
GRPON	O	0388	4	LASTVALUM	GM	0006	1
MOVE	OM	03A2	4	MOVER	OM	03CA	4
PAN	O	03F2	4	PLOT	OM	041A	4
PLOTR	OM	0448	4	READINGS[]	GM	010C	1
SIEVE	B	00012854	23	SIN	O	0668	6
STARR	B	00001842	17	TEXT17	B	00006E10	18
TICTOC	B	00012620	22	F\$rgmy	OM	04D2	4

Figure 5-2. Symbol Table

Common Symbol Table

The common symbol table lists all of the labeled and blank common regions assigned by XLL. Refer to Figure 5-3 for an example of the Common Symbol Table format.

An explanation of each column follows:

Common Name is the name assigned to the common region. In the case of blank common, the name is printed as "\$BLANK".

Origin is the address (in hexadecimal) assigned to the common region in the machine code segment.

Length is the number (in hexadecimal) of bytes reserved for the common region.

Extended BASIC Linking Loader (XLL) Version 1.0 05-Oct-84 14:17 Page 3		
Common Name	Origin	Length
ABC	17D4	0028
\$BLANK	17FC	0014

Figure 5-3. Common Symbol Table

Error Message Table

The fourth page lists error messages for all multi-defined or undefined symbols referenced by XLM. If symbol reference errors are found, this table will list the symbol name, the problem (whether the symbol is multi-defined or undefined), and the module in which the problem is found.

THE EXTENDED BASIC LIBRARY MANAGER PROGRAM

The Extended BASIC Library Manager program, XLM, creates and maintains software libraries. Software libraries are collections of standard and proven program modules that are used as part of other programs. XLM provides the capability to create libraries and to merge, delete, list, and copy modules in libraries. Extended BASIC libraries include a symbol dictionary to make efficient external symbol searches possible.

The Extended BASIC Library Manager program resides on a 5-1/4 inch floppy disk, distributed with the rest of the XBASIC system. It has the file name of XLM.FD2.

Executing the XLM Program

To start execution of the Extended BASIC Library Manager program from the FDOS command line (interpreter) prompt, type:

XLM<RETURN>

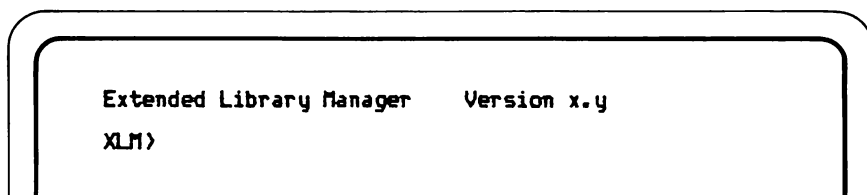
Terminating the XLM Program

Type /X or <CTRL>/Z at the end of a command line to exit from the Extended BASIC Library Manager program and return to FDOS. Type <CTRL>/P to exit the Extended BASIC Library Manager program at any other time.

Using the XLM Program

The Extended BASIC Library Manager program is used to create new libraries and to insert, delete, replace and copy modules within libraries. In addition, the Extended BASIC Library Manager program supplies listings of module names, and optionally, listings of the external references and definitions for each module.

When first started, the Extended BASIC Library Manager program displays:



```
Extended Library Manager    Version x.y
XLM>
```

NOTE

If the major version number (x) does not match the major version number on the front of this manual, call a Fluke Customer Service Center for advice.

Each command line is processed as soon as the <RETURN> key is pressed. The Library Manager performs the required functions and returns control to the user with the XLM> prompt. If errors are encountered, the command is ignored and an error message is printed. See Extended BASIC Library Manager Error Messages in the Error Messages section for complete descriptions.

XLM File Name Conventions

XLM, like XLL, permits the use of wild card characters to match input file names. These wild cards are the asterisk "*" that matches zero or more arbitrary characters in a file name or in an extension, and the question mark "?", that matches any one character in a file name or extension. File names that contain wild card characters are called patterns.

Examples of file name patterns, and the file names which they can match, are listed below. Arbitrary characters are indicated with an "x".

Pattern	Could Match
A*A	AA, AxA, AxxA, AxxxA, AxxxxA
A?A	AxA
A?*A	AxA, AxxA, AxxxA, AxxxxA
A?A*	AxA, AxAx, AxAxx, AxAxxx
A*B*A	ABA, AxBA, AxxBA, AxxxBA, ABxA, ABxxA, ABxxxA AxBxA, AxxBxA, AxBxxA, ...

With XLM, wild card characters may not be used in module names.

Extended BASIC Library Manager Commands

XLM commands create and maintain libraries of program modules. Commands are entered in response to the XLM> prompt.

COMMAND	DESCRIPTION
/C	Copy modules from a library to separate object files.
/D	Delete modules from a library.
/E	Extended list. Sends external symbol information to the screen or output file.
/L	List library module information to the screen or output file.
/M	Merge object files into a library.
/X	Exit and return to the FDOS> prompt.
?	Help. Lists the commands and gives an example for each.

Guidelines for these commands are:

- Both upper-case and lower-case entries are accepted. They are always equivalent.
- Only one command may be entered on a line.
- For each command (except exit and help), the general syntax is as follows:

[output]=input command

Two types of input files are recognized, object files and library files. Library files must be followed by opening and closing parentheses. A list of library modules may be listed within the parentheses. If the output file is not specified, the first input library name will be used, and the proper extension will be added.

- The syntax for file specifications (where required) is:

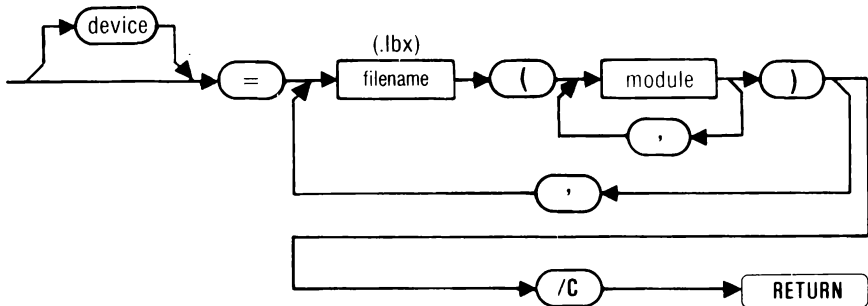
[device:]{filename} [.extension]

The extension is optional. If no extension is given for a library file, it is assumed to be .LBX. For object files and modules, the default is .OBX. The device is also optional and will default to SY0:, the system device. If a device is specified in the input section of a command, that device becomes the default for that command only.

/C

Copy

Syntax Diagram



Description

The Copy command reads the named object modules and copies them to individual files. If an output device is specified, the files are created on that device. Any output file specification is ignored. Modules may be taken from more than one library. If modules from two or more libraries have the same name, the most recent version is copied. Object files may not be given as input to a Copy command.

Examples

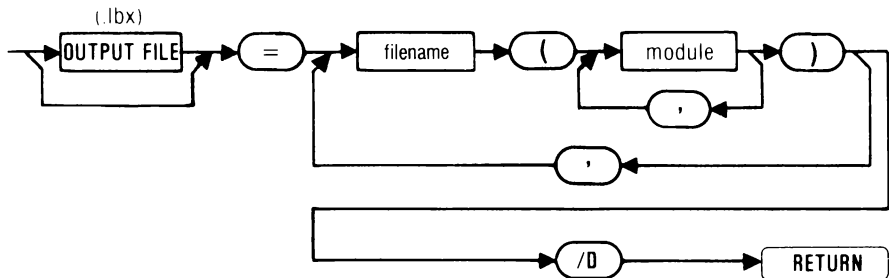
```
XLM)=lib1()/c
XLM)ed0:=lib2(mod1,mod2)/c
```

The first example copies all of the modules in lib1.lbx to individual files on the system default device (SY0:). The second copies the modules mod1.obx and mod2.obx from the library lib2.lbx on SY0: to individual files on ED0:.

/D

Delete

Syntax Diagram



Description

The Delete command removes modules from a library. More than one library may be given as input. Each input library must have at least one module listed. Object files may not be listed as input to a Delete command. If more than one library is given in the input list, the output library contains all modules from all libraries listed which have not been deleted.

Examples

```

XLM>outlib=lib1(mod6,mod2)/d
XLM>outlib=lib1(mod1,mod2), lib2(mod1)/d

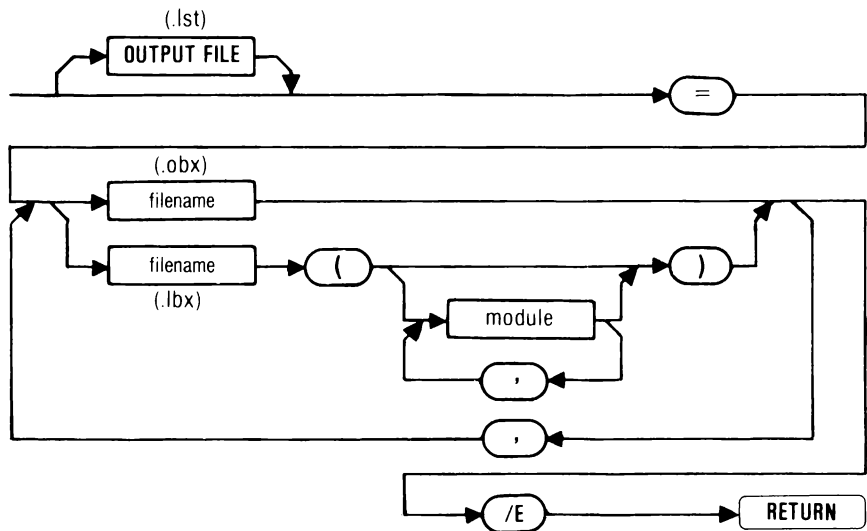
```

The first example takes all but two of the modules from lib1.lbx and makes a new library called outlib.lbx. The second copies parts of two libraries into outlib.lbx. If some of the modules from the two libraries have the same name, only the most recent are included in the output.

/E

Extended List

Syntax Diagram



Description

The Extended List command sends information about external symbols to the screen or a file. The output contains a list of external references and definitions for each module or object file. If no output file is specified, the information is printed on the standard output device.

Example

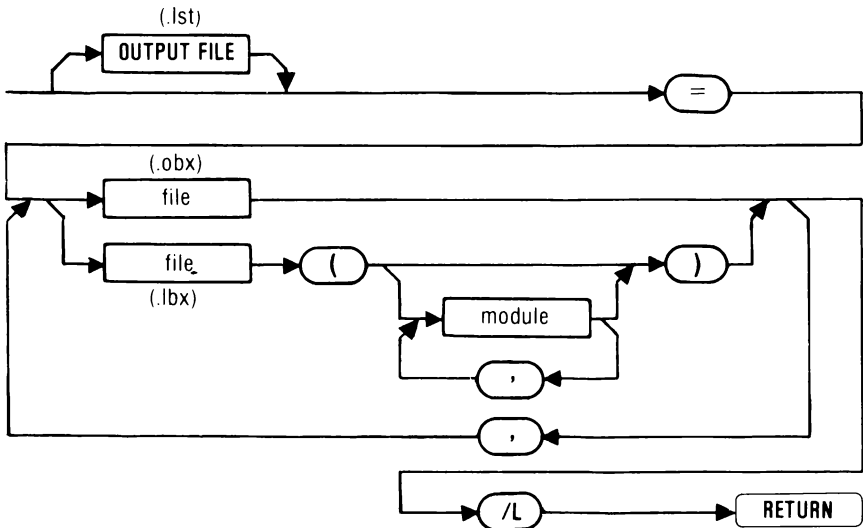
Here is a sample command and its corresponding output:

```
XLN)=q.lbx(assoc,key)/e
Library Map of SY0:Q.LBX on 12-Jul-84 at 14:17
ASSOC.OBX      02-Jul-84      08:55
  definitions:
    ASSOC
  references:
    F$XRIR      F$RGMY      F$DEV      I$CMDL      I$DEVL
KEY.OBX        03-Jul-84      10:06
  definitions:
    KEY
  references:
    F$RGMY      I$WORK
```

/L

List

Syntax Diagram



Description

The List command sends information about the modules in a library or object file to the screen or a file. If no output file is specified, the information is printed on the standard output device.

Example

Here is a sample command and its corresponding output:

```
XLN)=x.lbx(mod1,mod2)/1
```

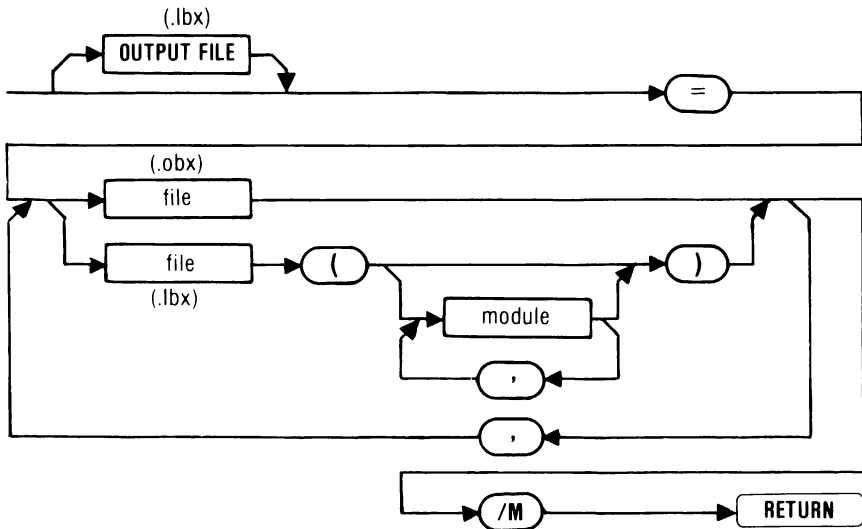
```
Library Map of SY0:X.LBX on 13-Jul-84 at 10:28
```

MOD1.OBX	05-Jul-84	09:15
MOD2.OBX	03-Jul-84	08:36

/M

Merge

Syntax Diagram



Description

The Merge command combines object files into a library. Existing library modules may also be included as input to the new library. If two modules appear with the same name, only the most recent module will be included in the output library.

Examples

```

XLM>outlib=ed0:mod1,mod2,inlib(mod5)/m
XLM>lib3=lib1(),mod7/m

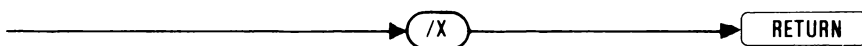
```

The first example combines mod1.obj, mod2.obj and a module from inlib.lbr into outlib.lbr. All three input files are from the device ED0:. The output file is stored on SY0:. The second example combines all modules in lib1.lbr with the object file mod7.obj and stores the output in the library lib3.lbr.

/X

Exit

Syntax Diagram



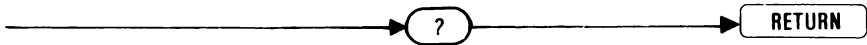
Description

The Exit command causes the Extended Library Manager to exit and return to FDOS. `/X` may be typed anywhere on a command line. All other inputs on the command line are ignored.

?

Help

Syntax Diagram



Description

The Help command prints a list of the commands and an example for each.

Example

When the Help command is used, the screen appears as follows.

XLM commands:		
Command	Function	Example
/C	Copy	MF0:=MATH()/C
/D	Delete	MATH.LBX=MATH.LBX(SIN,COS)/D
/E	Extended list	MATH.LST=MATH.LBX()/E
/L	List	MATH.LST=MATH.LBX()/L
/M	Merge	MATH.LBX=SIN.OBX,COS.OBX/M
/X	Exit	/X

Extended BASIC Library Manager Error Messages

When an error occurs, the Extended BASIC Library Manager program prints an appropriate message and returns to the XLM prompt. See the Error Messages Section for a list of XLM errors and their meanings.

Section 6

Extended BASIC Error Messages

CONTENTS

Introduction	6-3
Runtime and Compiler Errors	6-3
Extended Linking Loader Error Messages	6-8
Extended Library Manager Error Messages	6-15

INTRODUCTION

This section describes three types of error messages:

- Errors reported by the Extended BASIC Runtime system or Extended BASIC Compiler programs
- Extended Linking Loader error messages
- Extended Library Manager error messages

RUNTIME AND COMPILER ERRORS

The list of error codes that may be reported by the Extended BASIC Runtime System program or the Extended BASIC Compiler program is provided in Table 6-1. This list, rather than the error list in the Fluke Enhanced BASIC Programming Manual, should be used for reference.

Table 6-1. Fluke Extended BASIC Error List

CODE	LEVEL*	EXPLANATION
TYPE: OVERFLOW		
0	F	Memory overflow
1	F	Virtual array file > 64K bytes long, or > 64K elements (XBC)
2	F	Virtual array file too small for arrays
TYPE: SYSTEM		
100	F	BASIC interpreter or Runtime system internal error
101	F	Incompatible lexical file or Extended BASIC program
TYPE: COMMAND		
200	F	Immediate mode error
201	F	Cannot CONTINUE
202	F	STEP outside break mode
TYPE: I/O		
300	R	Device not Ready
301	R	Disk write protected
302	R	Illegal channel number specified
303	R	Channel already in use
304	R	Invalid device name or device not present
305	R	File not found on device
306	R	No room on device
307	R	Read/write past end of file
308	R	Channel not open
309	R	RS-232 channel input queue overflow
310	R	Input line too long
311	R	Disk read error
312	R	Illegal filename syntax
313	F	Random access to sequential file
314	F	Sequential access to random file
315	F	Virtual array assigned to sequential device
317	R	Illegal directory on device
318	R	Read (write) from (to) output (input) file
319	R	ON <channel> device not RS-232
320	F	Object file error
321	R	Device directory full
322	R	Illegal operation for device
323	R	File delete protected
324	R	Can't RENAME file
325	R	File medium swapped
326	R	Can't load - too little memory
327	R	Illegal image file format
328	R	Command line too long
329	R	RS-232 port number out of range
330	R	Parallel port number out of range

Table 6-1. Fluke Extended BASIC Error List (cont)

CODE	LEVEL*	EXPLANATION
TYPE: INSTRUMENT BUS CONTROL		
400	R	Illegal -488 port number
401	R	Illegal -488 device address
402	R	Illegal -488 secondary device address
403	R	Incomplete -488 handshake
404	R	Too many ports designated for -488 function
405	R	No devices attached to -488 port
406	R	No -488 ports available
407	R	-488 port specified is unavailable
408	R	-488 port timeout
409	R	Illegal WBYTE data
410	R	Parallel poll bit number out of range
411	R	Parallel poll bit sense not 0 or 1
412	R	-488 timeout limit out of range
413	R	TERM string longer than one character
414	R	No -488 driver in System
415	W	SET SRQ status byte value out of range
416	R	Illegal -488 operation for current port state
TYPE: SYNTAX		
500	F	Unrecognized statement
501	F	Illegal character terminating statement
502	F	Illegal subscript (<0)
503	F	Mismatched parentheses
504	F	Illegal let
505	F	Illegal if
506	F	Illegal line number
507	F	Illegal PRINT
508	F	Illegal format for PRINT or NUM\$()
509	F	Illegal INPUT statement
510	F	Illegal array dimension size
511	F	Badly formed define
512	F	Illegal FOR statement
513	F	FOR without NEXT
514	F	NEXT without FOR (jump back into "for" loop)
515	F	Unmatched quotes
516	F	Ill-formed expression
517	F	Bad OPEN statement
518	F	Bad CLOSE statement
519	F	IEEE-488 syntax error
520	F	Initial COM at illegal point in program
521	F	Not a well-structured statement
522	F	Illegal variable name
523	F	ON statement syntax error
524	F	OFF statement syntax error
525	F	TRACE syntax error
526	F	Illegal file size in open

Table 6-1. Fluke Extended BASIC Error List (cont)

CODE	LEVEL*	EXPLANATION
TYPE: SYNTAX (cont)		
527	F	RENumber parameter error
528	F	RENumber syntax error
529	F	ELSE without IF
530	F	NEXT syntax error
531	F	INPUT WBYTE requires IEEE-488 input
532	F	Illegal subrange descriptor
533	F	WBYTE/RBYTE data not integer type
534	F	Can't specify column for WBYTE/RBYTE subrange
535	F	Can't use undimensioned variable for WBYTE/RBYTE
536	F	Virtual array illegal for WBYTE/RBYTE
537	F	2-dimensional array illegal with WBYTE/RBYTE I/O
538	F	Illegal CONFIG statement
539	F	Illegal RBYTE syntax
540	F	RBYTE increment ≤ 0
541	F	Illegal RBYTE cycle length
542	F	Illegal WBYTE clause syntax
543	F	WBIN/RBIN precision error
544	F	WAIT statement syntax error
545	F	Illegal CALL statement
546	F	Virtual array parameter illegal
547	F	Parameter syntax error
548	F	Illegal SET statement syntax or option
549	F	Require file name for SAVE
550	F	Illegal RENAME statement syntax
TYPE: MATH		
600	F	Illegal mode mixing
601	R	Arithmetic overflow
602	R	Arithmetic underflow
603	R	Divide by zero
604	R	Square root argument > 0
605	R	Exponent too large
606	R	Log argument ≤ 0
607	R	Trig function argument too large
608	R	Illegal argument(s) for power operator
609	F	Illegal floating-point operation code
610	F	Unimplemented floating operation attempted

Table 6-1. Fluke Extended BASIC Error List (cont)

CODE	LEVEL*	EXPLANATION
TYPE: TRANSFER		
700	F	Illegal GOTO or GOSUB
701	F	RETURN without GOSUB
702	F	RESUME outside interrupt handler
703	F	CALL to undefined FN
704	R	ON expression GOTO selector out of range
705	F	CALL to undefined subroutine
706	F	Parameter count mismatch for CALL
707	R	Illegal time/date value
708	R	Timer value not initialized for ON INTERVAL or ON CLOCK
TYPE: INPUT		
800	R	Out of DATA in READ
801	W	Too much data entered for INPUT
802	W	Too little data entered for INPUT
803	W	Illegal character for INPUT or VAL()
804	F	Bad format in data statement
TYPE: VARIABLE		
900	F	Access to undefined variable
901	W	Redimension of array
902	R	Subscript out of range
903	F	COM of variable which is already defined
904	W	String too long for virtual array field
905	F	Incompatible COM declaration
906	F	DIM' within nested interrupt handler
907	F	Bad XOP 1 call
908	F	Illegal array parameter (memory vs. virtual)
909	F	Illegal conformal dimensioning parameter
* F = Fatal R = Recoverable W = Warning		

EXTENDED LINKING LOADER ERROR MESSAGES

Extended Linking Loader Error Messages may be divided into two types:

- General execution errors
- I/O errors

General execution errors, followed by their descriptions, are listed in Table 6-2. I/O error codes that may appear with XLL are listed in Table 6-3.

Table 6-2. Extended Linking Loader Error Messages

Error Message	Explanation
"{character}" is an illegal command character	The character described cannot appear in a command.
"{device}" is an invalid device name	The device given in the command cannot be a device name.
"{string}" is not a legal file name pattern	The pattern given is not a valid file name pattern.
"{string}" is not a valid command	The command given is not a command recognized by XLL.
Can't create file "{filename}" — {I/O error }	The named file cannot be created for the reason given.
Can't open input file "{filename}" — {I/O error }	The named file cannot be opened for reading for the reason given.
Can't use CBASIC symbol "{name}" in module {number }	The Extended BASIC subroutine named cannot be called from the Assembly or FORTRAN language subroutine whose module number is given.
Can't use file "{filename}" — Require file-structured device	Only file-structured devices may be used for input and output image files.

Table 6-2. Extended Linking Loader Error Messages (cont)

Error Message	Explanation
Can't write to file "<filename>" — File is protected	The named output file is protected against overwriting.
Comma or end-of-line must follow file name	Something other than a "," or the end of the line follows the name of a file in a command to XLL.
Directory error for device "<device name>": — <I/O error>	The device directory cannot be read for the reason given. This error may occur if input files are serial devices.
Duplicate MAIN program in file "<filename>"	More than one BASIC main program is present in the input file. The file names of the additional main programs are printed.
End of line expected after "end" or "go"	Only the end of the input line may appear after an END or GO command.
File "<filename>" has improper format	The input file named has a format that cannot be an object or library file.
File "<filename>" has no object modules	The input file named contains no object modules prior to the end of file mark.
File name(s) must follow command keyword	The FIND, INCLUDE, and OUTPUT commands require a file name following the command keyword.
FIND is useless prior to INCLUDE	If no INCLUDE commands precede a FIND command the libraries cannot resolve any undefined external references. No modules from the libraries can possibly be included.
Illegal object tag "<character>" in file "<filename>"	The character shown cannot appear in a .OBJ or .LIB file.

Table 6-2. Extended Linking Loader Error Messages (cont)

Error Message	Explanation
Internal program error: {module code}	The XLL program has detected an error in its own processing. Contact a Fluke Customer Service Center for advice.
Label "{name}": conflicting definition	The label shown has been defined as both a common region and a procedure name.
Library "{filename}" in illegal format	The library named has a format that cannot be a .LBX library.
Machine code segment is too large for memory!	The sizes of the machine code object modules plus the size of the BASIC global variable region exceeds 64K bytes, the address limit of the processor.
Memory overflow — cannot continue	The number of files, load modules, and symbols defined by the input to XLL exceeds the memory capacity of the processor.
No files match the pattern "{pattern}"	A pattern given with an INCLUDE or FIND command does not match any of the file names on the file storage device.
No MAIN program present	None of the BASIC program modules contains a main program.
No input file specified	An END or GO command was given before any INCLUDE files were specified.
No output file specified	An END or GO command was given before any OUTPUT file was specified.
No symbol table in library "{filename}"	No symbol table (external definition dictionary) could be found in the .LBX file named.

Table 6-2. Extended Linking Loader Error Messages (cont)

Error Message	Explanation
Object file error: Absolute loadbias address illegal, file <filename>	Absolute load addresses and load biases are not permitted in machine code object modules used with Extended BASIC.
Object file error: Bad symbol table attributes, file <filename>	The Extended BASIC module (from a .OBJ or .LBX file) has an illegal symbol table. This may be due to an error in the BASIC Compiler or in the Extended Library Manager. Contact a Fluke Customer Service Center.
Object file error: Call to non-subroutine, file <filename>	A CALL statement in a BASIC program refers to the name of a labeled COMMON region in a machine code module.
Object file error: Can't read object header, file <filename>	The initial record of an object module cannot be read. A preceding I/O error message should indicate the reason.
Object file error: Checksum error, file <filename>	An object module in a .OBJ or .LIB file contains invalid data.
Object file error: Entry address ignored, file <filename>	A machine code object module (.OBJ or .LIB file) contains an entry point definition. Execution of a program may only begin with the BASIC main program.
Object file error: Illegal (odd address) backchain, file <filename>	A machine code object module (.OBJ or .LIB file) contains an illegal external reference chain address.
Object file error: Illegal character in identifier: "<char>", file <filename>	The object file named probably contains invalid data; an external symbol has a non-printing character.
Object file error: Illegal hexadecimal data, file <filename>	A machine code object module (.OBJ or .LIB file) contains an illegal character where a hexadecimal number is required.

Table 6-2. Extended Linking Loader Error Messages (cont)

Error Message	Explanation
Object file error: Invalid \$DATA segment number, file <filename>	The \$DATA segment of a machine code module must be numbered zero.
Object file error: Invalid E-tag offset in module <number>, file <filename>	The index attached to an E-tag in the machine code module named exceeds the number of external references in the module.
Object file error: Invalid OBX tag type, file <filename>	An invalid object code tag was found in the .OBX file named.
Object file error: Invalid symbol number, file <filename>	An invalid symbol table reference was found in a .OBX module.
Object file error: Premature end of file, file <filename>	The physical end of the file was reached before the logical end of an object module was processed.
Object file error: Ref to undefined COMMON segment, file <filename>	A machine code object module made a reference to a COMMON segment that had not been defined.
Object file error: Too many COMMON segments, file <filename>	Only 127 COMMON segments may be referenced in any one machine code object module.
Object file error: Too many external references in module <number>, file <filename>	Only 1024 external symbols may occur in any machine code or compiled BASIC subroutine.
Only one output or map file name permitted	Only one file name may be specified with a MAP or OUTPUT command.
Output file too large: maximum is <number> blocks	The program is larger than the maximum output file size of 4 megabytes.

Table 6-2. Extended Linking Loader Error Messages (cont)

Error Message	Explanation
Output file too small - <number> blocks needed	The largest space that could be allocated on the output device is too small to hold the output image file.
Patterns illegal in output and map file names	Wild card file name characters cannot be used in output file names.
Patterns not permitted in device names	Wild card file name characters cannot be used in device names.
Read error in file "<filename>" — <I/O error>	An error occurred while trying to read the file named.
Symbol "<name>": conflicting definition in module <number>	The symbol named is defined both as a labeled COMMON region and as a program variable or entry point. Consult the load map for information on all of the modules that contain symbol definitions.
Symbol "<name>": duplicate definition in module <number>	The symbol named has been defined more than once. Consult the load map for information about on all of the modules that contain symbol definitions.
Symbol "<name>" is undefined	The symbol named is referenced but never defined. Consult the load map for information on the module or modules making reference to the symbol.
The OUTPUT command requires a file name	The OUTPUT command must be followed by the name of the output file to be created.
Unable to load overlay — <I/O error>	The XLL program consists of four overlays. If an overlay cannot be loaded, the program prints this message and exits to FDOS. Be sure that XLL's loading device is operating and that the disk, if any, is not swapped during XLL's execution.
Write error in file "<filename>" — <I/O error>	The output or map file cannot be written to for the reason given.

Table 6-3. Extended Linking Loader I/O Error Messages

I/O ERROR MESSAGE	EXPLANATION
Device does not exist	The device is unknown to the operating system.
Device hardware error	The device is inoperable or the file medium is faulty.
Device not ready	The device is not powered up or does not contain a disk.
Directory overflow	No room exists in the device directory for another file name.
File does not exist	The file name could not be found in the device's file directory.
File is protected	The file's directory entry prohibits overwriting or deleting the file.
File medium swapped	The disk in a disk drive was either changed or removed and re-inserted while files were being accessed by XLL.
Illegal device directory	The device directory is in illegal format.
Illegal device or file name	A device or file name has invalid syntax or characters.
Illegal operation for device	An operation was attempted that is illegal for the device. (For example, reading the directory of a serial channel.)
Medium write protected	The device or disk cannot be written to.
No end-of-file character	A .LIB or .OBJ file does not end with an end of file character.
No FDOS driver for device	There is no FDOS module that knows how to work with the device named in the error message.
No memory for I/O buffer	XLL does not have enough free memory left to create an I/O buffer area for this file.
No room on device	No room exists on the device to permit the creation of another file.
Read/write past physical end of file	For an output file: the file is not large enough. For an input file: the file is incomplete or has been truncated.

EXTENDED LIBRARY MANAGER ERROR MESSAGES

When an error occurs, the Extended BASIC Library Manager program prints an appropriate message and returns to the XLM prompt. Table 6-4 lists these messages and a description of each.

Table 6-4. Extended Library Manager Error Messages

ERROR MESSAGE	EXPLANATION
---------------	-------------

An = must follow the output file

An equal sign must separate the output specification from the input list.

Can't create {object file}

The copy command (/C) could not successfully create the named object file.

Can't create temporary file

The delete (/D) or merge (/M) command could not successfully create the temporary file.

Can't open library {library name}

The named library can't be found or read.

Can't open object file {file name}

The named object file can't be found or read.

Can't use a serial device for that command

An attempt was made to use a serial device as input or output for a copy, delete, or merge command.

Commands: /c, /d, /e, /l, /m, /x or ?

A non-existent command was entered.

Device error

A non-recoverable error was detected during transfer to or from the floppy or electronic disk.

Device not ready

The disk is not inserted, or the disk drive door is open.

Table 6-4. Extended Library Manager Error Messages (cont)

ERROR MESSAGE	EXPLANATION
End input list with command	No command was given, or illegal characters are present in the input list.
End of library hit unexpectedly	End of file was reached while searching through an existing library. The library should be recreated.
Fatal errors - library not altered	A new library was not created due to errors encountered during the generation of the temporary library.
File name too long	File name is longer than 15 characters.
First input library can't be a pattern	When using the delete or merge commands, the first input must not be a pattern.
If no output is specified, the first input must be a library	In a delete or merge command, the first input must be a library when no output is specified before the equal sign. The default output is generated using this library name.
Illegal characters in input list	Characters that are not allowed in file names are included in the input list.
Illegal characters in module list	Characters that are not allowed in file names are included in the module list.
Incorrectly formed symbol in module {name}	The module named has a symbol that is not ended with null bytes.

Table 6-4. Extended Library Manager Error Messages (cont)

ERROR MESSAGE	EXPLANATION
Library {library name} does not contain {module name}	The module specified as a member of the library listed is not found in the library.
Library module not in proper format	A library module is not in the proper format. The library should be recreated.
Library not in proper format	An input library is not in the proper format. The wrong file may have been accidentally specified.
Multiply defined symbol: {symbol name} in modules {module 1} and {module 2}	The modules listed both define the given symbol.
No files match the pattern {pattern}	No file on the specified device matches the pattern given.
No such device	The device specified does not exist.
Object file not in proper format	An object file is not in the proper format. File may not actually be an object file.
Object files not allowed with delete and copy commands	Use FUP to delete or copy object files.
Patterns not allowed in module list	Patterns are only allowed in library and object file names.
Start with a file name or /X to exit	The command line must start with the output file, an equal sign if no output file is specified, or an /X.
Use FUP to delete a whole library	No modules were specified in a delete command.

Section 7

Data Storage

CONTENTS

Introduction	7-3
Overview	7-3
Using Arrays for Data Storage	7-4
Array Types and Differences	7-5
Advantages and Disadvantages	7-6
I/O Channels	7-7
Channel Input and Output	7-7
OPENing an Output Channel	7-8
OPENing an Input Channel	7-8
Close a Channel	7-8
Specifying File SIZE	7-9
Sequential Data Files	7-10
Creating a Sequential File	7-10
Reading a Sequential File	7-11
Creating Main Memory Arrays	7-13
Using Main Memory Arrays	7-14
Using Main Memory Arrays as Ordinary Variables	7-14
Programming Techniques	7-15
Two Dimensional Arrays	7-16
Multiple Arrays	7-17
Redimensioning	7-17
Main Memory Arrays and Program Chaining	7-18
Serial Storage of Main Memory Arrays in Mass Storage .	7-19
Device Storage Size Requirements	7-20
Device File Size Calculation	7-20
Disk Size Calculation Example	7-21

CONTENTS, *continued*

Virtual Array Files	7-22
Definition	7-24
Advantages and Disadvantages	7-24
File Structure	7-26
Analogy	7-26
Virtual Array File Organization	7-27
Storage Size Requirements	7-29
Virtual Array Size Calculation	7-30
Creating Virtual Arrays	7-32
Using Virtual Arrays	7-33
Using Virtual Arrays as Ordinary Variables	7-33
Using Virtual Array Strings	7-34
Programming Techniques	7-35
Array Element Access	7-35
Splitting Arrays Among Files	7-36
Reusing Virtual Array Declarations	7-37
Equivalent Virtual Arrays	7-38

INTRODUCTION

Many program applications require data to be stored or retrieved from some form of long term storage. For instance, a digital voltmeter connected to the IEEE-488 bus does not have the capability to average its readings over a long period of time. The solution is to take ten (or however many) readings, store them, then use the math capability of the Controller to compute the average.

Using simple variables to store the readings is effective, but addressing these variables systematically in a program can be clumsy. A better technique is to store the data in an array. This allows addressing via the array subscript, which can be a numeric expression.

OVERVIEW

This section describes the techniques of data storage, both in main memory and on file-structured devices. After discussing the advantages and disadvantages of both, the methods of use and the statements used are discussed.

USING ARRAYS FOR DATA STORAGE

An array is both a powerful and convenient method of storing a large volume of data. There are 954 possible names for each type of simple variable in Fluke BASIC. This may seem like a large number, but it is not enough to store the readings from a typical digital voltmeter measurement. Consider the problem of storing 5 readings using sequential, but independent variables using a loop.

```

10 FOR X = 1 TO 5
20   GOSUB 100      ! subroutine to request data (yX)
30   IF X = 1 THEN A1X = YX
40   IF X = 2 THEN A2X = YX
50   IF X = 3 THEN A3X = YX
...
80 NEXT X
90 END
100 ! subroutine to get reading from dvm
110 ! more code, return value in yX
120 RETURN

```

This program fragment will work, but it is easy to see how involved and clumsy it could get, especially if a large number of data items were involved. Notice that each reading consumes a program line and one variable name. Here is a program fragment using an array to accomplish the same thing.

```

10 DIM AX(6X)
20 FOR X = 1 TO 5
30   GOSUB 100 ! go get reading (yX)
40   AX(X) = YX
50 NEXT X
60 END
100 ! subroutine to get reading from dvm
110 ! more code, return value in yX
120 RETURN

```

In line 40, the array element is assigned the value returned from the subroutine (Y%) at line 100. The next iteration through the loop increments the value of X%, which also causes line 40 to assign Y% to the next sequential array element.

If the number of readings to be made and stored were increased to 1000, the only program changes needed would be to lines 10 and 20. In the first example 995 more program lines would need to be added inside of the FOR-NEXT loop.

Array Types and Differences

Fluke BASIC allows data values to be stored by one of the following methods.

- As a sequentially-accessed data file on a file-structured device. This is essentially a list of data items.
- In main memory. Main memory arrays may be one or two-dimensional.
- As a virtual array (random access data file) on a file-structured device. The array may be one or two-dimensional.

Advantages and Disadvantages

Each of the three storage methods has its strong and weak points. These are summarized below.

- The sequential data file and virtual array are both stored on file-structured devices. If the device is a non-volatile file-structured device (such as the floppy disk or bubble memory), the array storage is also non-volatile.
- A main memory array, being stored in main memory (system RAM), is susceptible to power failures, program chaining and DELETE ALL statements.
- A sequential data file must be accessed sequentially. If you want the 405th element of the file, you must sequentially access the preceeding 404 elements first, which can be time consuming.
- Sequential data files must be written, then read. A sequential file may not be simultaneous read/write.
- Virtual arrays and main memory arrays are random access. Any array element may be accessed at any time.
- An array stored in main memory can have a shorter access time than the same array stored on a file-structured device.
- All three storage methods will accept integer, string or floating-point (real) values.
- The elements of a virtual string array have a definite, dimensioned length. Elements of a main memory string array are limited in length only by the amount of main memory available.
- Virtual arrays can have up to 65,536 (64K) bytes of data (128 blocks).
- Main memory arrays are limited to 28K bytes minus the program size in K bytes. Assuming a 5K application program, this means a 23K maximum array size.
- Virtual arrays and main memory arrays can be used like ordinary program variables.

I/O CHANNELS

The Instrument Controller communicates between the BASIC program and various devices by means of I/O channels.

The *devices* used for data storage are *file-structured* devices such as:

MF0: (floppy disk)
ED0: (E-disk)
WD0: (fixed disk drive)
MB0: (bubble memory)

There can be a maximum of 16 I/O channels open at any one time. They are designated with the numbers 1 through 16.

The I/O channels are also used to communicate with instruments connected to the IEEE-488 bus and RS-232 devices. Refer to sections 9 and 10 of this manual.

Channel Input and Output

The BASIC statements used in conjunction with the I/O channels are the PRINT statement and the INPUT statement.

- The PRINT statement is used to output data from the program to the device via the I/O channel.
- The INPUT statement is used to input data from the device, via the I/O channel to the program.
- The #n clause is added to both the PRINT and INPUT statements when they are used for channel I/O. The #n clause specifies the channel number to be used for input or output.
- The PRINT and INPUT statements are fully described in the Reference Section of this manual.

OPENing an Output Channel

To designate an output channel, it is necessary to OPEN it AS a NEW FILE. When an output channel is to be used for a virtual array, the DIM clause is used in the OPEN statement.

Example

```
10 OPEN "MFO: EXAMPL.DAT" AS NEW FILE 1 ! new file, channel 1
20 OPEN "EDO: TEST.DAT" AS NEW FILE 2 ! new file, channel 2
30 OPEN "MFO: TEST.VRT" AS NEW DIM FILE 3 ! virtual array, chan 3
30 PRINT #1, "HELLO" ! output "HELLO" to channel 1
40 PRINT #2, A$ ! output A$ to channel 2
```

NOTE

*A virtual array also requires the use of the DIM statement.
Refer to the discussion of Virtual Arrays, elsewhere in this section.*

OPENing an Input Channel

To designate an input channel, it is necessary to OPEN it AS an OLD FILE.

Example

```
10 OPEN "MFO: EXAMPL.DAT" AS OLD FILE 3 ! input from channel 3
20 OPEN "EDO: TEST.DAT" AS OLD FILE 4 ! input from channel 4
30 INPUT #3, A$ ! read first line, chan 3
40 INPUT #4, B$ ! read first line, chan 4
```

An error will result if an attempt is made to OPEN a channel that has previously been OPENed but not CLOSED. It is a good practice to close channels at the end of a program unless files are passed to a chained program.

CLOSE a Channel

The CLOSE statement closes the file associated with the channel number given. It is a good idea to CLOSE a channel immediately prior to OPENing that channel.

Example

```
10 CLOSE 1
20 CLOSE 2
```

Specifying File SIZE

If no file size is specified, the largest contiguous file space will be temporarily reserved for that file. This may not leave any device space available for additional files. To overcome this, specify the SIZE in blocks that you wish reserved on the device for that file. Methods for calculating block size are described in the discussions of Main Memory Arrays and Virtual Arrays.

Example

```
20 OPEN "MFO: FILE3.DAT" AS NEW FILE 1 SIZE 2  
30 OPEN "MFO: FILE4.DAT" AS NEW FILE 2 SIZE 6
```

SEQUENTIAL DATA FILES

A sequential data file is a list of data items, separated by (CR) (LF). As its name implies, a sequential file must be read sequentially; there is no means to access an item within the file except by reading all the items that precede it.

Sequential files store data in ASCII format and may be read using the COPY statement or FUP. To copy a sequential file to the screen, type the following command from BASIC:

COPY {pathname}

or, from FDOS:

FUP {pathname}

Creating a Sequential File

Creating a sequential file is best illustrated with a short program. The discussion that follows describes the important points to remember.

```
10  CLOSE 1      ! be sure channel is closed before opening
20  Y% = 1%
30  OPEN "TEST.DAT" AS NEW FILE 1% !test.dat, new file, chan 1
40  FOR XX = 1 TO 5%
50  PRINT #1, Y%  ! output y% to channel 1
60  Y% = 2 * Y%   ! do something to y%
70  NEXT XX
80  CLOSE 1
90  END
```

The example program opens a new file named "TEST.DAT", then writes the value of Y% to the file. The FOR-NEXT loop causes this to repeat 5 times.

Line 10 closes the channel to avoid a possible error if the channel had been opened previously and not closed.

Line 30 opens the file "TEST.DAT" and assigns channel 1 for data transfers to the file. The NEW clause tells BASIC to open a NEW file to store data. If the file "TEST.DAT" exists already, it is overwritten.

The PRINT statement within the FOR-NEXT loop sends the value of Y% to channel 1 (and hence to the file).

Line 80 CLOSEs channel 1 (good housekeeping).

Reading a Sequential File

Reading a sequential file is illustrated here with three examples. The discussion that follows each program, and the comments within the programs emphasize the important points.

Example 1

The COPY statement provides a very simple method of reading a sequential file. It is so simple, however, that it is not possible to direct the output anywhere else but to the display.

```
10 COPY "TEST.DAT"
```

Example 2

```
10  CLOSE 1      ! insurance
20  ! look at the file generated previously
30  OPEN "TEST.DAT" AS OLD FILE 1
40  ON ERROR GOTO 100 ! use this to detect end-of-file
50  INPUT #1, AX      ! load AX with data from file
60  PRINT AX          ! show it!
70  GOTO 50           ! repeat
100 IF ERR = 307 THEN 110 ELSE 130 ! err 307 = eof
110 PRINT "END OF FILE REACHED"
120 GOTO 150
130 PRINT "ERROR "; ERR; "OCCURED" ! other error than eof
150 CLOSE 1 \ END
```

The example program opens the file created by the previous example program, then reads each line of the file, prints the value, then loops until the end-of-file is reached. When the end-of-file is reached, the error handler at line 100 prints an appropriate message.

Line 10 ensures that the channel to be used in line 30 is not already open. An error will result if it is.

Line 30 opens the file "TEST.DAT" for reading (OLD clause) and assigns it to channel 1 for data transfer.

Line 50 assigns the first line of the file to a%. Line 60 prints a% on the display.

Line 70 causes an infinite (until an error) loop.

The error handler at line 100 prints the message at line 110 if the error was caused by reaching the end-of-file, otherwise a brief (and somewhat cryptic) error message is printed.

Line 150 closes the previously opened channel/file and exits.

Example 3

The INCHAR function may also be used to read data from a sequential data file. The INCHAR function is described in the Reference volume of this manual set.

```
10 CLOSE 1
20 OPEN "TEST.DAT" AS OLD FILE 1
30 CX = INCHAR(1)
40 IF CX = 26 THEN 70
50 PRINT CHR$(CX)
60 GOTO 30
70 CLOSE 1 \ END
```

insurance
open file
read character
check for end of file
print it
loop to do it again
clean up, go home

Line 10 ensures that the channel has been closed before an attempt is made to open it at line 20.

Line 20 opens the file "test.dat" and specifies that it shall be read (as opposed to written) and associates it with I/O channel #1.

Line 30 uses the INCHAR function to read one character from the previously opened channel.

Line 40 checks C% for an end of file character and branches to line 70 if C% is an end of file character.

CREATING MAIN MEMORY ARRAYS

A main memory array is simple to create and use. Use the following steps to create a main memory array.

1. Dimension the array using the DIM or COM statement. The following statement dimensions a 100 element single-dimensioned integer array using N1% as the array variable. The use of the COM statement is described later in this section.

```
10 DIM N1X(100X)
```

2. Assign values to the array elements (in any sequence) by using the LET or INPUT statements.
3. Array element values may be used using the LET or PRINT (USING) statements or by using the array element as part of a numeric or logical expression.

USING MAIN MEMORY ARRAYS

The following paragraphs present some suggestions for using main memory arrays.

Using Main Memory Arrays as Ordinary Variables

Except for the required DIM (or COM) statement prior to use, a main memory array can be treated as an ordinary variable. That is, data may be stored or retrieved from any array element at any time, in any sequence. The COM statement is discussed later in this section, Section 13 of this manual, and in the Reference volume of this manual set.

- Any legal variable name may be an array variable.
- Remember to use the DIM or COM statement first.
- Do not use the DIM or COM statement twice on the same array.
- If your program crashes, data stored in a main memory array is lost after execution of a RUN or EDIT statement unless the COM statement was used to allocate memory space for the array.

In the following example, elements of A% may be used wherever integer array elements may be used.

```
10      DIM AX(255X)
```

The following example shows that data may be read from or written to the array simply by writing the array name in an expression or assigning a value to an array element.

```
305     IF AX(IX) > 0X THEN 350
350     ! more code
430     LET AX(IX) = ABS (AX(IX)) + 2X
```

Programming Techniques

The program given in the introduction to this section illustrates the “nuts-and-bolts” of getting values into a main memory array. Getting values out of the array in a sequential fashion is largely the same:

1. Set up a FOR-NEXT loop.
2. Read the array elements one-by-one using the LET or PRINT statements.
3. Repeat step 2 as necessary.

Another method is to use array subranging. This is a special case of the array element identifier. The first example displays all elements of the array A\$, which has previously been defined as having 20 elements in line format. The second example displays only elements 5 through 10 in columnar format (note the semicolon).

```
10 PRINT A$(0.:19)  
20 PRINT A$(5.:10);
```

The example program in Section 7 of the System Guide demonstrates this method also. Look in the “Transfer Module” portion of the program. The semicolon following the array subrange suppresses the normal (CR) (LF) after each element, allowing display in multi-columnar format.

Two Dimensional Arrays

Until now, only one dimensional arrays have been used and discussed. It may help to think of a one-dimensional array as a one row matrix. For example, the preceding example array, A\$, represented as a matrix would be:

← Columns 0 through 19 →

ROW A\$(0) A\$(1) A\$(2) A\$(3) A\$(4) A\$(5) A\$(6) A\$(7) A\$(8) A\$(9) ... A\$(19)

Fluke BASIC allows two-dimensional arrays. A two-dimensional array such as A\$(1,4) has two subscripts in its array element identifier. 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$(2,4)
20 READ PS$(0,0),0,0,0,0,0
30 SHIFT DAY DAY DAY DAY DAY
40 1 1 2 3 4 5
50 DATA 10, 12, 9, 7, 10
60 DATA 9, 14, 10, 11, 11
70 DATA 11, 7, 6, 8, 10
80
90 FOR IX = 0 TO 2
100 PRINT PS$(IX,0..4)
110 PRINT
120 NEXT IX
130 END
140
```

! dimension a 3 X 5 array
! read 15 values

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. 0	Col. 1	Col. 2	Col. 3	Col. 4
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. 0)	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, eg.:

```
10  DIM AX(3,5), BX(4,10), AS(10,100)
20  DIM RL(1000)
30  DIM AS(10,10)
:
```

Redimensioning

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.

Main Memory Arrays and Program Chaining

A main memory array must have been created with the COM statement to survive program chaining. The COM statement replaces the DIM statement for that array. The COM statement reserves variables and arrays in a common area for reference by chained programs.

- Only real (floating-point) and integer variables may be used with the COM statement.
- String variables may not be stored in the common area.
- Use a virtual array for string variables that must be accessed by chained programs.
- All programs accessing a common area must use COM statements that are identical in order, type, and array sizes; the actual variable names, however, may be different.
- The contents of the common area are lost when the EXEC, EXIT, or DELETE ALL statements are executed.

For example, assume that a chained program requires the use of three floating point simple variables, an integer simple variable, a floating point array, and an integer array defined in a previous program. The first program could use a COM statement such as:

```
10      ! Program A
20      COM A, B, C, FX, D(24X), TX(100X)
      .
      .
1050     RUN "B"
1060     END                                ! End of program A
```

The second program could then use:

```
10      ! Program B
20      COM L1, L2, L3, GX, K(24X), PX(100X)
      .
      .
```

Note that while the names of the variables stored in the common area have changed between programs, the order and type of the variables are exactly the same.

Serial Storage of Main Memory Arrays In Mass Storage

A main memory array may be stored and retrieved from a mass storage device. Two methods are possible:

- Store the array in a sequential data file.
- Store the array in a virtual array.

The following examples illustrate how array data can be serially stored and retrieved from the disk (MF0:). 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, floating-point, or string) and dimensions.

The first example program stores the letters "A" through "F" in a main memory array, then writes that array to a sequential data file. The second example program retrieves that array from the data file, then prints the array.

- The array variable used in the second example could be any legal string variable. The data file holds only the data with no clue to the identity of the variable used to store the data.

Data Storage:

```
10  ! "EXAM1.BAS"
100 CLOSE 1
110 OPEN "EXAM1.DAT" AS NEW FILE 1 SIZE 1 ! INITIALIZE
! "NEW" in line 110 indicates new file created on the disk. ! RESERVE & NAME DISK SPACE
115 DIM A$(5X) ! DECLARE 5 ELEMENT ARRAY
120 FOR I% = 0% TO 5%
130   A$(I%) = CHR$(65% + I%) ! ASSIGN LETTERS A --> F
140 NEXT I%
150 PRINT #1, A$(0%..5%) ! STORE ON DISK
160 CLOSE 1 ! CLOSE FILE
170 END
```

Data Retrieval:

```
10  ! "EXAM2.BAS"
100 CLOSE 1
110 OPEN "EXAM1.DAT" AS OLD FILE 1 ! INITIALIZE
! "OLD" in line 110 indicates file already exists on disk ! OPEN FILE, ASSIGN CHANNEL
115 DIM A$(5X)
120 INPUT LINE #1, A$(0%..5%) ! INPUT FILE DATA TO ARRAY
130
140 CLOSE 1 ! CLOSE FILE
150 PRINT A$(0%..5%) ! DISPLAY DATA FROM ARRAY
160
170 END
```

Device Storage Size Requirements

Before a main memory array can be stored, space must be reserved for it on the file-structured device. When a new file is OPENed, the largest available contiguous space on the storage medium (floppy disk or other file-structured device) 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 device, BASIC 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

Device File 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 (floating-point), INTEGER, and STRING elements you plan to use in the array file. Array values are stored on the file-structured device as ASCII values. One byte of device storage is required for each character stored.

Device 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, 3X	requires 5 bytes
PRINT #1, -3X	requires 5 bytes
PRINT #1, USING "8*", -3X	requires 4 bytes
PRINT #1, 3.285	requires 9 bytes
PRINT #1, -3.285	requires 9 bytes
PRINT #1, USING "8*.***", -3.285	requires 8 bytes
PRINT #1, "12345"	requires 7 bytes

Disk Size Calculation Example

Calculate the SIZE requirement for the DIM statement in the following program:

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

Assumptions:

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

String Integer Real EOF

$$\begin{aligned}\text{BYTE SIZE} &= 100 [(10+2) + (4+2) + (7+2)] + 1 \\ &= 100 [27] + 1 \\ &= 2701\end{aligned}$$

$$\text{BLOCK SIZE} = 2701 / 512 = 5.275391$$

Since partial blocks are not allowed; the next highest integer = 6 Blocks.

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

VIRTUAL ARRAY FILES

A virtual array provides the BASIC programmer with an easy-to-use, non-volatile means of program data storage. Once created, a virtual array variable may be treated like any other BASIC variable. A virtual array is bidirectional; you may read or write from it any time, in any sequence.

A virtual array may be assigned values from a main memory array and vice versa; virtual arrays can be used in equations with main memory arrays. Virtual arrays are different from main memory arrays in the following ways:

1. Main memory arrays reside in main memory. Virtual array elements temporarily reside in a main memory buffer of 512 bytes (1 block) per channel (file) number. They permanently reside on a file-structured storage medium. Virtual arrays may also reside on E-disk (expansion RAM memory), however, E-disk storage is volatile.
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 non-volatile file-structured storage medium (CLOSEing the file ensures this).
3. Virtual arrays are not initialized by the DIM statement. Main memory arrays are assigned initial values by the DIM statement.
4. 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.
5. Virtual arrays can be made equivalent, (two arrays can share the same area of file memory).
6. 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.

7. 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.
8. Since main memory arrays must share main memory with the BASIC program, the maximum amount of main memory available for main memory arrays is limited to 28K bytes, minus the size of the user program in K bytes.
9. A virtual array file can be as large as 65,536 bytes. The total number of virtual array files is limited by the available file-structured storage.
10. Virtual arrays are stored as binary data and cannot be viewed by FUP.
11. 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 (crash).
12. Virtual array data survives a re-RUN or EDIT of the program, but main memory array data is lost in both of these situations.

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 one of the file-structured storage media. 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 additional considerations in some situations due to their fixed length.

Definition

A virtual array is a collection of data stored in a random access file-structured storage device, such as the floppy disk or bubble memory. The data is stored in the Instrument Controller's 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.

Advantages:

1. Non-volatile ---survives power down of the Controller; survives program chaining and the DELETE ALL statement.

NOTE

A virtual array is non-volatile only if the file-structured device on which it is stored is non-volatile.

2. Virtual arrays do not "consume" main memory, thus more program space is available.
3. Random access means PRINT and INPUT statements are not necessary for data I/O.
4. Supports equivalencing.
5. String data, in addition to numeric data, can be accessed by chained programs.
6. Text messages can be stored on the storage medium rather than in main memory. The same text can be used as often as needed.
7. The program can be restarted after a Controller power down and returned to the exact place in the program where execution ceased (due to powering down). (See note after point #1, above.)

8. Virtual arrays can be many times larger than main memory arrays; up to 16 times (over 1024K bytes) as much data can exist in virtual arrays when a floppy disk, E-disk, or Winchester disk are used.

Disadvantages:

1. Virtual arrays are not allowed in RBYTE or WBYTE statements.
2. Virtual arrays execute slower 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 and device. For example, the E-disk is almost as fast as main memory.
3. Unlike main memory arrays where the DIM statement assigns a 0 value to floating-point (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. To guard against bogus data entering your program, you may wish to initialize the entire array to some known value prior to storing data in it.

File Structure

Virtual arrays must reside on a specific physical space on the file-structured device in order for the program to know where it can go to store and retrieve data. An analogy to the disk file structure is a helpful aid to understanding where virtual arrays are stored.

Analogy

Imagine you have a storage room (disk) which contains 80 file cabinets (tracks) and each cabinet has 10 file drawers (blocks). Each file drawer contains 512 folders (bytes). The terms “Blocks” and “Sectors” are used interchangeably.

The file name for a specific storage space for virtual arrays always appears on the front of a drawer, i.e., a file can never begin in the middle of a drawer (block). Likewise, an array element cannot overlap from one drawer to the next (all the bytes for an element must exist in the same drawer (block)). A specific file name can use as many as 128 drawers (blocks).

The disk directory contains the track and sector (block) associated with each file name. This directory is consulted by the operating system (FDOS) each time a file name is referenced by the program. File names are associated with a channel number (file no.). To make our analogy complete, let us further imagine that the file room (containing the 80 cabinets) is located in a very large building and you (the program) are located away from the storage room and it is actually possible for you to take any one of sixteen different routes (channels 1 through 16) to reach the storage room. Before you store data or retrieve data from a file drawer (name) it is necessary to specify which route (channel no.) will be used to transport the data. As long as a file name has a drawer OPEN, the designated route (channel no.) cannot be used for another file name until the channel is CLOSEd, and the next file OPENed using the same channel number.

Finally, imagine that you have a utility cart (buffer) to transport one file drawer (512 byte block) back and forth between you (main memory), and the disk (storeroom).

In actuality, the contents are never removed from a disk block. Instead, its contents are copied onto the main memory buffer and when the buffer is sent back to update the disk, the buffer contents are copied back into the file on the disk. The temporary file is the same size as the entire permanent file.

Virtual Array File Organization

When a virtual array file is opened, BASIC creates a 512-byte (1 block) buffer in main memory to hold the block of the file currently being used (one buffer is created for each virtual array file). Each file is considered to consist of a sequence of bytes, numbered 0 (first block) to n (last block). The description of each virtual array contains the channel number to which the file containing the array is attached, and the address within the file at which the array starts (the array's base address).

The base address for a virtual array is determined when the DIM statement declaring the array is processed. The base address assigned is the next available (higher) address which will not cause an array element to cross a block boundary. Each array element must be wholly contained within a 512-byte block. This restriction may be defined as: the base address of an array must be an integral multiple of the array element length and no array element may be longer than 512 bytes. This works since all virtual array elements have a length which is an integral power of 2.

Since virtual arrays are assigned addresses in the file in the order in which the arrays are declared in the DIM statement, the restrictions noted in the paragraph above suggest that it is possible to allocate file space efficiently or inefficiently when arrays having differing element lengths are assigned to the same file. This depends on the order in which array declarations appear in the DIM statement. To eliminate wasted file space, the simplest rule is that virtual array declarations should appear in the DIM statement (as read from left to right) in decreasing order of array element lengths. This rule ensures that if an element overlaps a block boundary, a minimum of space is left unused in the previous block.

NOTE

The unused space at the end of a virtual array file is available for use if a subsequent DIM statement enlarges the file. See the discussion that follows on equivalent virtual arrays.

In the following DIM statement, the arrays are allocated space as shown below the statement.

DIM #1%, A\$ (10%), B (10%, 9%), C% (1%, 4%)

A\$	11 elements of 64 bytes each	= 704 bytes
B	110 elements of 8 bytes each	= 880 bytes
C%	10 elements of 2 bytes each	= 20 bytes
TOTAL:		1604 bytes

The total space needed is 68 bytes greater than 3 blocks (1604 - (3 * 512)). The blocks will be allocated as follows. Note that the first three blocks are completely used, leaving only the extra 68 bytes for the fourth block. The 444 bytes remaining in the fourth block are unused.

BLOCK	VARIABLE	ELEMENTS	BYTES
1	A\$	8	512
2	A\$	3	192
2	B	40	320
3	B	64	512
4	B	6	48
4	C%	10	20

Suppose the DIM statement is changed to read:

DIM #1X, CX (1X, 4X), B (10X, 9X), A\$ (10X) = 64X

The total space needed remains 1604 bytes. The variables however, are allocated to blocks as follows.

BLOCK	VARIABLE	ELEMENTS	BYTES
1	C%	10	20
1	unused	—	4
1	B	61	488
2	B	49	392
2	unused	—	56
2	A\$	1	64
3	A\$	8	512
4	A\$	2	128

Only 508 bytes of block 1 and 456 bytes of block 2 are used. The unused portions, totalling 60 bytes, could not entirely contain one more data element in the sequence assigned. As a result, block 4 has only 384 bytes available, instead of the possible 444.

Storage Size Requirements

Before a virtual array can be dimensioned, space must be reserved for it on the disk. When a new virtual array file is OPENed, the largest available contiguous space on the file-structured storage medium is allocated for the single file, unless the SIZE is included in the OPEN statement. If the NEW files are OPENed without a SIZE statement and there is only one contiguous space available on the device, BASIC 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.

Virtual Array SIZE Calculation

SIZE must be stated as an integer number of BLOCKS (1 BLOCK = 512 BYTES). A virtual 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 floating-point (real) elements, integer elements and string elements you plan to use in the virtual array file. Each floating-point element occupies 8 bytes. each integer element occupies 2 bytes. Each string element occupies 1 byte/character. Elements are not allowed to overlap block boundaries. If an element is too large to fit in the remaining storage space in a block, the remaining space is left vacant and the element is placed in the next higher block. To eliminate wasted file space, the simplest rule is that virtual array declarations should appear in the DIM statement from left to right in decreasing order of array element lengths.

Appendix J lists a program which allocates virtual arrays by blocks according to your DIM statement.

String elements require special consideration since all elements for a given string array variable name must be the same number of characters in length *and* that length must be 2 or 4 or 8 or 16 or 32 or 64 or 128 or 256 or 512 characters, i.e., any power of 2 between 2 and 512 inclusive. String element length is assigned in the DIM statement. If no length is specified, the default length is 16 characters.

Based on the above considerations, and given:

R = the No. of REAL elements in all FLOATING-POINT arrays

I = the No. of INTEGER elements in all INTEGER arrays

S = the No. of STRING characters in all STRING arrays

V = vacant bytes prior to boundaries of occupied blocks

the formula becomes:

$$\text{SIZE} = (R * 8 + I * 2 + S + V) \text{ BYTES} / 512 \text{ BYTES} / \text{BLOCK}$$

Example

DIM #1, A(10), AA(5,6), BX(30), BB%(3,7), AS(10) = 8, AA\$(50,50) = 2

Calculate the SIZE for the following virtual array dimension statement:

$$\begin{aligned}
 R &= 11 + (6 * 7) &&= 53 \text{ elements for A(10) and AA(5,6)} \\
 I &= 31 + (4 * 8) &&= 63 \text{ elements for B\%(30) and BB\%(3,7)} \\
 S &= 11 * 8 + (51 * 51 * 2) = 5290 \text{ characters for AS(10) and} \\
 &&&\text{AA\$(50,50)}
 \end{aligned}$$

$$V = 0$$

Block Allocation Table

424 Bytes	88 Bytes	38 Bytes	80 Bytes	394 Bytes	4608 Bytes	288 Bytes	224 Bytes
53 Reals	44 Integers	19 Integers	10 8 Char. Strings	197 2 Char. Strings	2304 2 Char. Strings	144 2 Char. Strings	224 Vacant Bytes
Block 1		Block 2			Blocks 3 thru 11	Block 12	

NOTE

Remember element numbering starts with 0, e.g., DIM #1, A(10) yields 11 elements; DIM #1, A(0) yields one element.

$$\text{SIZE} = (53 * 8 + 63 * 2 + 5290) / 512 = 11.40625 \text{ BLOCKS}$$

Only whole blocks may be allocated, i.e., SIZE must be an integer, the next higher whole number is the correct answer: SIZE = 12 BLOCKS.

CREATING VIRTUAL ARRAYS

Creating virtual arrays requires the following actions:

1. A filename must be associated with the virtual arrays. (OPEN {filename} AS... statement.)
2. A channel number must be associated with the filename. (FILE n clause of OPEN... statement.)
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). (AS {NEW, OLD} clause of OPEN statement.)
4. The SIZE of the arrays in blocks should be stated. (SIZE clause of OPEN statement.)
5. The arrays must be DIMensioned. (DIM statement)
6. The DIMension must be associated with the channel number picked in step 2, above. (#n clause of DIM statement.)
7. The channel (file) must be CLOSEd in order to transport the most current array data (contained in the buffer) to the disk.

The following example program illustrates the process of creating a virtual array.

```
10 CLOSE 1          ! insurance
15 ! new file, "test.vrt", virtual array, chan 1, size = 1 block
20 OPEN "TEST.VRT" AS NEW DIM FILE 1 SIZE 1
30 DIM #1, A(5X)      ! dimension array a, 5 elements, chan 1
40 FOR IX = 0 TO 4X    ! loop to load array
50 A(IX) = IX          ! assign value
60 NEXT I             ! loop
70 PRINT A(0..4)      ! print array
80 CLOSE 1           ! close channel/file
90 END
```

USING VIRTUAL ARRAYS

Once a virtual array file has been opened and a DIM statement for the channel has been executed, the virtual array elements may be used just as ordinary variables.

Using Virtual Arrays as Ordinary Variables

In the following example, elements of A% may be used wherever integer array elements may be used, except in RBYTE, WBYTE, or CALL statements. (See the section on IEEE-488 Bus Input and Output Statements.)

```
10      OPEN "INTEGR.BIN" AS DIM FILE 1%
20      DIM #1%, AX(255%, 127%)
```

The following example shows that data may be read from or written to the file simply by writing the array name in an expression or assigning a value to an array element.

```
305     IF AX(1%, J%) > 0% THEN 350
430     LET AX(K%, 0%) = ABS (AX (K%, 1%)) + 2%
```

Using Virtual Array Strings

Strings in virtual arrays are considered by BASIC to be of fixed length. The default length is 16 characters, or as declared in the DIM statement (see the DIM discussion in this section).

The following example specifies a virtual string array with character elements. Line 390 will display the number 32 regardless of the value assigned to that particular element of A\$.

```
380     DIM #2X, A$(15X) = 32X
390     PRINT LEN(A$(1X))
```

When characters are assigned to a virtual array string element, BASIC will add null characters to the right end of the string until it equals the declared string element length. This can be the source of subtle errors. Consider the virtual array A\$ of the previous example. The following program section attempts to add an * character to all of the elements of A\$. This example will not work, and results in error 904 (string too long for virtual array string field).

```
370     FOR IX = 0X TO 15X
380     A$(IX) = A$(IX) + "*"
390     NEXT IX
```

Each element of A\$ is allocated 32 characters. The expression A\$(IX) + "*" results in a 33-character string. When this string is assigned to A\$(IX), error 904 (string too long for virtual array string field) results. It is necessary to strip trailing null bytes from the virtual array string value before appending the '*' character.

The TRIM statement causes trailing null bytes to be removed from a virtual array string. The format of the TRIM statement is:

```
TRIM tX
```

where t% specifies whether or not to trim trailing null bytes from a virtual array string. If t% is zero, no trimming is performed. If t% is not zero then all trailing null bytes are trimmed.

The TRIM statement must be issued before reading the first virtual array string element for which null byte trimming is desired.

PROGRAMMING TECHNIQUES

The key to the efficient use of virtual arrays is to minimize the number of data transfers to or from the virtual array file. Whenever a virtual array element is accessed, either to read its value or to assign to it a new value, BASIC determines the block number and the file in which the element exists. If the block required is not in the memory buffer, the required block is moved from the file into the memory buffer. The block previously held in the buffer is written to the file only if a change in its contents occurred.

Array Element Access

Array elements in a file are stored in row-major order which means that access to the elements, in the storage order, is most efficient when the rightmost array subscript varies most quickly, as in the following example:

```
4650   FOR IX = 0% TO 63%  
4660     FOR JX = 0% TO 63%  
4670       AX (IX, JX) = 0%  
4680     NEXT JX  
4690   NEXT IX
```

The form just discussed describes the most efficient access method for initializing the array A%. When the array A% is stored on a floppy disk, its initialization required 7.05 seconds. When the same array is stored on the E-disk, its initialization required 4.71 seconds. The following example is the least efficient access method:

```
4650   FOR IX = 0% TO 63%  
4660     FOR JX = 0% TO 63%  
4670       AX (JX, IX) = 0%  
4680     NEXT JX  
4690   NEXT IX
```

This second example requires 254.33 seconds (8.05 seconds on ED0:), 36 times as long (1.7 times for ED0:) as the first example. The first example requires a new block to be read for every 256 elements of A% that are written. This second example, however, requires that a new block be read for every four elements of A% that are written.

Splitting Arrays Among Files

If information stored in different virtual arrays will often be required at the same time, placing the arrays in separate files will speed processing. The following example illustrates the value of utilizing separate files for two parallel virtual arrays as opposed to placing both arrays within the same file.

Example

In some programs it may not be possible to use the null stripping function described previously in this section. This may be because the null characters are part of the string data required. However, the true string length can be determined without stripping the null characters. If there is no character which may be used to specify the end of a string in the virtual array, this information may be retained by placing an integer array parallel to the string array. Thus, for each string element, an integer element contains the length of the string.

In the following example, `L%(I%)` contains the length of string `A$(I%)`. The significant characters of element `A$(I%)` can be recovered by a statement such as line 6370.

```
6365 OPEN "DATA.VRT" AS DIM FILE 1%  
6366 DIM #1%, A$(1023%) = 16%, L%(1023%)  
6370 B% = LEFT (A$(I%), L%(I%))
```

A drawback to this method is that each time an element of `A$` is read from the beginning of the file, another block of the file must be read to retrieve the corresponding element of `L%`.

A more efficient organization is to assign `A$` to one virtual array file and `L%` to another file. This will cause the BASIC Interpreter to assign two buffers, one for the strings of `A$` and one for the integers of `L%`.

```
4785 OPEN "DATA1.VRT" AS DIM FILE 1%  
4790 OPEN "DATA2.VRT" AS DIM FILE 2%  
4770 DIM #1%, A$(1023%) = 16%  
4780 DIM #2%, L%(1023%)
```

In an actual test, accessing the elements of `A$`, one by one in increasing order, the first example (with `A$` and `L%` in one file) required 386 seconds. With `A$` and `L%` in separate files, only 15.8 seconds were required to perform that same processing, a 24:1 difference.

Reusing Virtual Array Declarations

When a virtual array DIM statement has been executed, the variables defined as virtual arrays remain defined even after the virtual array file has been closed. However, an attempt to access a virtual array when the channel has been closed will result in an error 308 (channel not open). Since the variables remain defined, it is possible to close a virtual array file and to later re-open it. Since the variables have already been defined, a new DIM statement is not necessary to re-open the file. Note, however, that the file must be re-opened using the same channel number as that used in the DIM statement defining the arrays.

If a number of separate virtual array files, all with the same data organization, must be processed by a program, it is possible (if no two files are to be processed simultaneously) to reuse the variable definitions. Consider the following processing sequence:

OPEN	first file
DIM	virtual arrays
	process first file
CLOSE	first file
OPEN	second file (on same channel as 1st file)
	process 2nd file
CLOSE	second file
.	
.	
.	

In each processing loop, the same virtual array variables may be used to process the file data.

Equivalent Virtual Arrays

It is possible to execute multiple DIM statements for a single channel. The second (and subsequent) DIM statements for a channel simply redefine the file organization as shown below. The first DIM statement allocates a 441-element array, A%, organized as a 21x21 matrix. The second DIM statement does not allocate array B% following A% but redefines the 441 integer elements of the file as a vector with the name B% (similar to a FORTRAN equivalence).

```
110    DIM #1%, A% (20%, 20%)  
120    DIM #1%, B% (440%)
```

NOTE

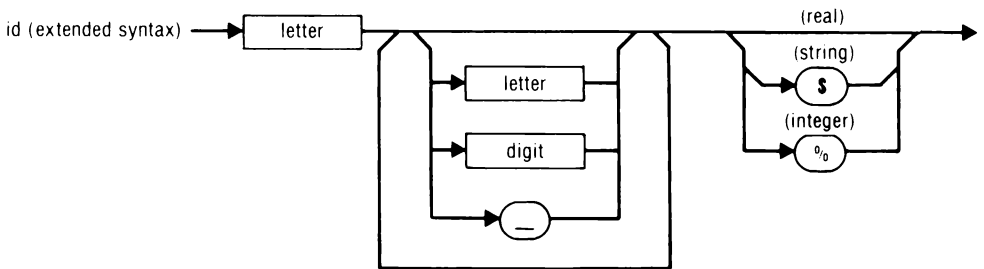
Fluke BASIC makes no check for the consistency of virtual array equivalences. This is the responsibility of the programmer.

Appendices

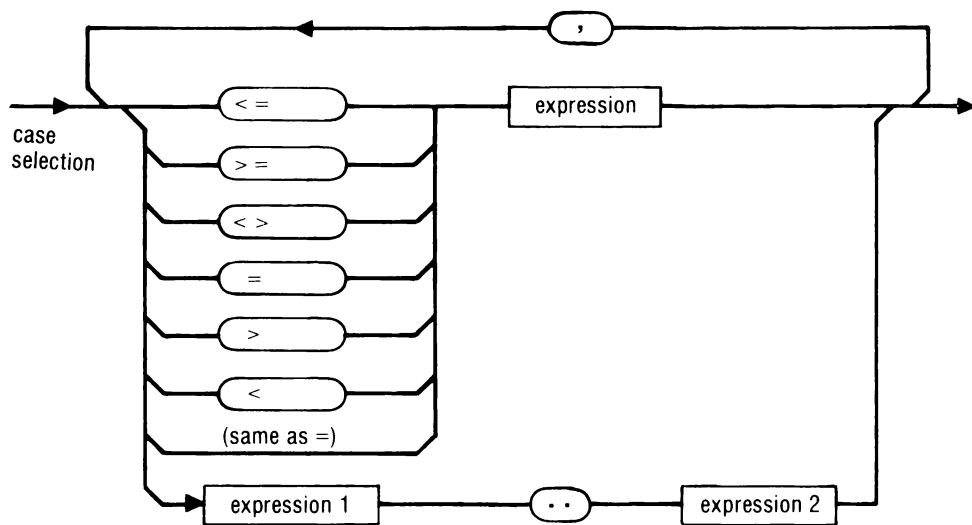
CONTENTS

A	Supplementary Syntax Terminology Diagrams	A-1
B	ASCII/IEEE Bus Codes	B-1
C	Extended BASIC Language Software	C-1
D	Reserved Words	D-1
E	Integrating Subroutines from Other Languages	E-1
F	Glossary	F-1

Appendix A



Supplementary Syntax Diagrams



Appendix B

ASCII/IEEE Bus Codes

Appendix C

Extended BASIC Language Software

This appendix describes the programs supplied with the 1722A-203 Extended BASIC Language option. Each program is listed along with a short description followed by the name of the reference manual that describes the software. Any special installation or operational requirements are also listed here.

- | | |
|--|--|
| XBC.FD2 | Extended BASIC Compiler program. This program is used to translate the Extended BASIC source program into a machine-language file. The Extended BASIC Compiler program is described in the Extended BASIC Manual. |
| <p>The Extended BASIC Compiler is an overlay-structured program. The BASIC Compiler requires 64K of user space to execute. If the Compiler resides on a floppy disk, the disk may not be removed while XBC is running.</p> | |
| XLL.FD2 | Extended Linking Loader program. The Extended Linking Loader program is a simple linkage editor that is used to create executable programs from compiled files. Refer to the Linkage Utilities Section in the Extended BASIC Manual. |
| XLM.FD2 | Extended Library Manager program. The Extended Library Manager program is used to create and maintain libraries of Extended BASIC program modules and subroutines for use with Extended BASIC programs. Refer to the Linkage Utilities section in the Extended BASIC Manual. |

BSXRUN.FD2

Extended BASIC Runtime System program. The Runtime System program converts the Instrument Controller into a machine that runs Extended BASIC programs. Refer to Section 3 of the Extended BASIC Manual.

The Extended BASIC Runtime System program must be run whenever Extended BASIC programs are used on the Instrument Controller. This program must be on a file-structured device whenever a Extended BASIC program is run. The Extended BASIC Runtime system program requires 64K of user space for execution. The Runtime system program is described in Section 4 of the Extended BASIC Manual.

FTN\$IF.LIB

FORTTRAN Interface Runtime Library. This is a library of FORTRAN subroutines that replace the FORTRAN Runtime System program when FORTRAN subroutines are used from compiled BASIC programs. In addition, the FORTRAN Interface Runtime Library contains subroutines to be used in FORTRAN subroutines for exchanging strings and error codes between FORTRAN subroutines and compiled BASIC programs. Refer to Appendix E of this manual.

XBCC.CMD

Extended BASIC Command File. This is a command file that automatically performs the sequence of commands to create simple (not overlay-structured) Extended BASIC programs.

XBCE.CMD

Extended BASIC Command File. This command file is similar to XBCC.CMD. Use this command file for Extended BASIC programs that use the Extended Syntax option (/E).

Appendix D

Reserved Words

Table D-1 lists reserved words that conflict with BASIC language statements. These reserved words may NOT be used in Extended BASIC as

- Variable name
- Statement label names
- Literal subroutine names (in most cases)

The reserved words may be used as subroutine names with the following restrictions:

- Implied CALL statements may not be used with subroutines using a reserved name. For example, the implied CALL statement to a subroutine named STOP

STOP

will execute a BASIC STOP function. Use the statement

CALL STOP

to branch to the subroutine.

- The name “FN” may not be used for any other purpose than a user-defined function. All strings beginning with FN are considered functions.

Reserved Words

Table D-2. Reserved Words

ABS	ECHO	LCASE\$	QDIR	TAB
ALL	EDIR	LEAVE	RAD\$	TAN
AND	ELSE	LEFT	RANDOMIZE	TERM
AS	ELSIF	LEN	RBIN	THEN
ASCII	ENABLE	LET	RBYTE	TIME
ASH	END	LINE	READ	TIME\$
ASSIGN	ENDIF	LN	REM	TIMEOUT
ATN	ENDLOOP	LOCAL	REMOTE	TO
BREAK	ENDSELECT	LOCKOUT	RENAME	TRACE
	ENDWHILE	LOG	REPEAT	TRIG
CALL	ERL	LOOP	RESTORE	TRIM
CASE	ERR	LSH	RESUME	UCASE\$
CHR\$	ERR\$	MEM	RETURN	UNPROTECT
CLEAR	ERROR	MID	RIGHT	UNTIL
CLOCK	EXEC	MOD	RND	USING
CLOSE	EXIT	NEW	RUN	VAL
CMDFILE	EXP	NEXT	SELECT	WAIT
CMDLINE\$	EXPORT	NOECHO	SET	WBIN
COM	FILE	NOT	SGN	WBYTE
CONFIG	FLEN	NUM\$	SHELL	WHILE
COPY	FN	OFF	SIN	WITH
COS	FOR	OLD	SIZE	XOR
CPOS	GOSUB	ON	SPACE\$	
CTRL/C	GOTO	OPEN	SPL	
DATA	IF	OR	SQR	
DATE	IMPORT	PACK	SRQ	
DATE\$	INCHAR	PASSCONTROL	STEP	
DEF	INCOUNT	PI	STIME\$	
DIM	INIT	PORT	STOP	
DIR	INPUT	PORTSTATUS	SUB	
DISABLE	INSTR	PPL	SUBEND	
DUPL\$	INT	PPOL	SUBRET	
	INTERVAL	PPORT		
	KEY	PRINT		
	KILL	PROTECT		

Appendix E

Integrating Subroutines From Other Languages

INTRODUCTION

This appendix to the Extended BASIC Programming Manual describes some factors to be considered when designing a program that is composed of elements written in different programming languages.

Extended BASIC programs are composed of a main program segment that is written in the Extended BASIC language and optionally, subroutines that are written in the Extended BASIC, Compiled BASIC, FORTRAN, and Assembly languages. When using a mixture of subroutines from different source languages, a programmer must be aware of two potential problems: passing incompatible data types and illegal use of conformal dimensioning.

NOTE

Extended BASIC subroutines may not be used unless the main program is also written in Extended BASIC (so that the Extended BASIC Runtime program will be running).

CAUTION

Unlike CBASIC subroutines, XBASIC subroutines may not be called by FORTRAN or Assembly language programs. If this is attempted, the Extended BASIC Linking Loader (XLL) will report an error when the program is linked.

PARAMETER DATA TYPES

Not all programming languages use the same data types. When subroutines written in different languages are mixed in one program, you must take care to make sure that parameters that are passed between program segments are a type that is available in the language in which the other program is written. Table E-1 shows the data types that are compatible between programs written in Extended BASIC (and Compiled and Interpreted BASIC) and programs written in FORTRAN. While specific data types are not part of Assembly language programs, any of the data types used by programs written in Extended BASIC or FORTRAN may be used at the discretion of the programmer. Refer to the FORTRAN Programmer's Reference Manual or the BASIC Manual for information about the specific FORTRAN and BASIC data formats for use in Assembly language programs.

Table E-1. Compatible Data Types

EXTENDED BASIC	FORTRAN
Real	Double-Precision Floating-Point (REAL * 8)
Integer	Simple Integer (INTERGER * 2)
String	<not used>
<not used>	Real (Single-Precision Floating-Point)
<not used>	Complex
Integer	Logical
<not used>	Fixed

The following example shows how a variable must be defined as a double-precision floating-point type before being passed back to an Extended BASIC program as a Real number.

Extended BASIC:

```
...  
CALL FORSUB( COUNTER )  
...
```

FORTRAN:

```
SUBROUTINE FORSUB( COUNT )  
DOUBLE PRECISION COUNT  
...  
RETURN  
END
```

The Extended BASIC floating-point variable COUNTER is passed to the FORTRAN subroutine FORSUB which declares the double-precision variable COUNT. COUNT is used by FORSUB to accept and return the processed value of the Extended BASIC variable COUNTER.

DIMENSIONING

Multiply-dimensioned arrays become disorganized when exchanging dimensioned arrays between program segments that are written in different languages.

Following is an example of a call to a FORTRAN subroutine. In this instance, an array needs to be redimensioned to fit the characteristics of the language in which the program segment is written. This code segment illustrates a call to a subroutine that is written in FORTRAN from a program that is written in Extended BASIC.

```
10 !  
20 ! Extended BASIC main program  
30 !  
40 DIM A(100)  
100 CALL FORTRN(A(), 101Z)  
300 END
```

The program calls a FORTRAN subroutine named FORTRN and passes two parameters to it: the array A and the size of the array. The array was previously dimensioned to 100. Because BASIC arrays begin with element 0, the array contains 101 elements numbered 0 through 100.

The subroutine FORTRN is:

```
C  
C FORTRAN SUBROUTINE USED WITH EXTENDED BASIC  
C  
SUBROUTINE FORTRN(A, I)  
DOUBLE PRECISION A(I)  
...  
RETURN  
END
```

The subroutine FORTRN is defined in the first program line after the heading with two parameters: A (which was A() in the program that called this subroutine) and I (which was 101%).

Multidimensional Arrays

A conflict arises when passing arrays with more than one dimension between FORTRAN subroutines and BASIC program segments because of the different way that arrays are handled in each language. Arrays are stored in different sequences, which means that array elements are not numbered in the same way. Figure E-1 shows a sample two-dimensional array as it would appear in both BASIC and FORTRAN.

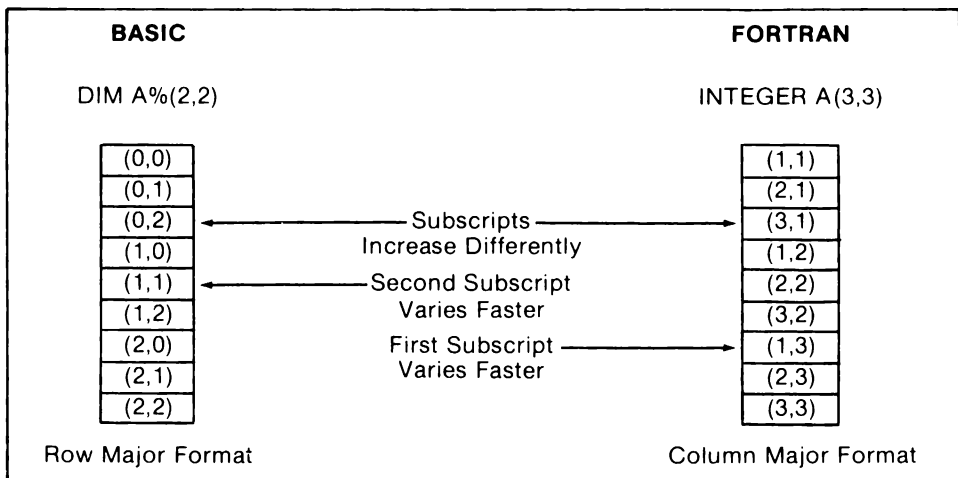


Figure E-1. Array Formats

Note that the sequence of element storage is different. In the example shown in Figure E-1, a value which is stored as (0,2) in a BASIC program would be element (3,1) if that array was passed to a FORTRAN subroutine. The confusion which results between the two formats makes passing multidimensional arrays between languages undesirable.

Appendix F

Glossary

This Appendix is a supplementary glossary of terms that occur in this manual. Use this Appendix with Appendix M in the Fluke BASIC Manual.

COMMAND-LINE OPTION

An addition to the command line of a program that configures the program to handle special events. For example, the BASIC Compiler program must be configured so that it will correctly compile programs that use Extended BASIC language options. These command-line options consist of a slash character (/) followed by various letters. The command-line below directs the BASIC Compiler program to compile a program named TEST that uses the optional extended syntax.

```
=TEST/E
```

Command-line options are sometimes known as switches.

COMPILER

A program that translates a source program written in a high-level programming language into a machine-coded object file. The resulting file is normally linked with other files and converted into an executable program using the Linking Loader or Linkage Editor programs.

COMPILED LANGUAGE

A language where source programs are completely translated into machine code, then executed in a separate step. This is in contrast to an Interpreted Language, where source statements are translated and executed one at a time.

CONFORMAL DIMENSIONING

A method of transferring attributes of a DIMensioned variable (number of dimensions and dimension size) to a true subroutine.

CONTINUED LINE

An Extended BASIC program line that contains imbedded <RETURN> characters to allow it to be used by utilities with limited line width capabilities or to impart a visual structure to the line. Imbedded <RETURN>s are preceded by an ampersand character (&) to distinguish them from <RETURN> characters at the end of a program line.

EXTENDED BASIC

A compiled BASIC language, similar to Compiled BASIC, which takes advantage of the extended memory of the Fluke 1722A Instrument Controller.

EXTENDED SYNTAX

An optional syntax provided by Extended BASIC to allow the programmer some freedom in structuring program lines and creating variable and subroutine names. The main features of the optional extended syntax are the use of continued lines, long variable names, true subroutines, and improved block structured control flow. The /E switch must be used to inform the BASIC Compiler program that a source program uses extended syntax.

GLOBAL VARIABLE

A variable that may be shared between modules in an Extended BASIC program. Global variables are defined and used via the EXPORT and IMPORT statements.

INTERPRETED BASIC

The standard version of the BASIC language. The BASIC Interpreter program translates a BASIC program into machine-executable code one line at a time while the program is in progress. It is the closest version to standard ANSI BASIC.

KEYWORD

A word that has a special meaning as part of a BASIC language statement.

LABELS

Labels are special words that may be placed at the beginning of a statement to identify that statement with a name. Thereafter, statements that normally refer to line numbers, such as GOTO and RESTORE, may refer to this label instead.

LIBRARY

A collection of modules that is maintained in a common file. The files in a library usually have related functions (for example, files collected together to form a matrix math library). In Compiled BASIC, libraries are arranged sequentially in the library so that there are no backward references. In Extended BASIC, libraries are arranged randomly.

LIBRARY MANAGER

A machine language program that is used to create and maintain libraries of program modules. The Extended Library Manager, XLM, replaces the LM program used with the Compiled BASIC, FORTRAN, and Assembly languages. It operates in much the same way as the LM program does, but creates random, rather than sequential libraries.

LINKING LOADER

A program that links Extended BASIC programs. The Extended Linking Loader, XLL, combines program sections into a main program segment, combines subroutines with the main program, and adjusts memory references in the program. The Runtime System is enabled and input and output files are created. XLL replaces the LL and LE programs used with the Compiled BASIC, FORTRAN, and Assembly languages.

LOAD MAP

Detailed listing of the modules that comprise an executable program, produced by the Extended Linking Loader Program, XLL, under control of the MAP command. The load map is stored in a print image file.

LOCAL SUBROUTINES

Subroutines that are entered using a GOSUB command and exited with a RETURN command. They are created and compiled as part of the same file as the program segment (either a main program or a true subroutine) that uses them.

MODULES

Object file program segments, either sections of main program bodies or subroutines.

OBJECT FILE

A file consisting of machine-executable information. It is normally the output of a compiler program. The object file usually needs to be linked with other program modules and loaded into memory before executing.

OPTION

A language feature that is available, but need not be used. Any time that an option is used, the BASIC Compiler program must be configured, via the command line, to compile the optional language feature properly.

OVERLAY

A subroutine that shares memory space with other parts of the program. A large program may be structured with overlays to use less memory space. Program sections are loaded into memory as needed.

PARAMETERS

Variables or constants that are exchanged between true subroutines and their calling routines are called parameters.

PASS BY REFERENCE

This is the parameter-passing convention used by Extended and Compiled BASIC and FORTRAN. The address of the parameter is passed to the subroutine so that any changes made to the parameter in the subroutine changes the value of the variable with which the subroutine was called (they are the same data item). Pass by Reference is also known as Pass by Address.

In the following example, subroutine SUB1 actually changes the value of variable A.

```
A = 1
CALL SUB1(A)
! A = 2...SUB1 CHANGES THE VALUE OF A
....
SUB SUB1(B)
B = 2
SUBEND
```

PASS BY VALUE

This parameter-passing convention passes the value of the parameter to the subroutine rather than to the address. Thus, changes to the parameter within the subroutine do not affect the variable with which the subroutine was called.

When an expression or constant is the parameter to a subroutine, it acts as if it were passed by value. Extended BASIC first copies the value of the expression or constant to a temporary location, then uses pass by reference to pass the address of the temporary location to the subroutine, rather than the address of the original variable. This protects the variable from changes in value caused by the subroutine.

```
CALL SUB1( (A) )
! A = 1...SUB1 USES THE VALUE A WITHOUT CHANGING IT
....
SUB SUB1(B)
B = 2
SUBEND
```

PHYSICAL LINE

The 80-character width of the Instrument Controller's display.

PROGRAM LINE

A complete BASIC line, ending with a <RETURN> character. It may consist of several continued lines.

RELOCATION

The process of creating a new copy of a program that will load into memory in a different place than a previous copy.

RESERVED WORD

A word that may not be used as a variable name, subroutine name, or label name because it conflicts with a BASIC statement or operator.

RUNTIME

Operations that occur or program functions that are used while a program is running, as opposed to operations that occur during compiling or linking.

RUNTIME SYSTEM

A program that makes it possible for a machine to run Extended BASIC programs. The output format of the Extended BASIC Compiler is an intermediate code rather than machine code. At runtime, the Runtime System program translates this intermediate code into machine recognizable statements.

SOURCE PROGRAM

The original program, whether it is written in a high-level language like FORTRAN or BASIC, or in a low-level language like Assembly. A source program needs to be translated into a machine-executable form before use.

STATEMENT LABEL

A name that identifies a program line, in the same way that a line number does. The label may be used in place of a line number in any branch instruction.

TRUE SUBROUTINES

Subroutines that may be created and compiled separately from a calling routine, that may exchange parameters with a calling routine, and may contain local variables that are not accessible from other parts of the program. True subroutines are defined by the SUB and SUBEND statements that bracket them. They are entered via CALL statements and exited via SUBEND or SUBRET statements.

r refers to a statement number in the BASIC Reference manual

ANSI standard BASIC, 2-5

Arrays, 2-19, **r3**

conformal dimensioning, 2-26, **E-4**

multidimensional, **E-5**

redimensioning, 2-28

redimensioning virtual arrays, 2-28

use in Subroutines, 2-25

variable, 2-26, 2-28

ASCII/IEEE-488 bus codes, **B-1**

Automatic integer conversion, 2-18

BASIC compiler, 3-3

BASIC compiler errors, 3-19

BASIC compiler options, 3-17

Branch statements, 2-32, 2-33

BSXRUN, 4-3, 4-4

<CTRL>/C, 4-7, 2-10 **<CTRL>/Z**, 4-7

CALL statement, 2-21, **r10**

Case (SELECT statement), 2-45, **r130a**

COM statement, 2-24, **r16**

Command files, 1-18, 3-11

Command line option, 3-12, 3-17

Compiled BASIC (CBASIC), 2-3, 2-4

Compiled language, 1-3, 2-4

Compiler

command format, 3-13

command line options, 3-4, 3-12, 3-17

configuration switches, 3-9, 3-19

default filename extensions, 3-13, 3-14, 3-25

errors, 3-19, 6-3

exiting, 3-16

installation, 3-9

options, 3-17

temporary files, 3-14, 3-15, 3-16

Compiling

into Object Code, 3-4

long programs, 3-18

programs, 1-3, 1-11

XBASIC program, 3-4

Conformal dimensioning, 2-26, **E-1**

CONT TO command, 2-53, **r18**

Continuation lines in extended syntax, 2-11, 2-15

Control flow statements, 2-40

Conventions used in this manual, 2-7

notation, 2-10

statement descriptions, 2-21

syntax, 2-5, 2-8

COPY command, 5-30

Copying a program using T-copy, 1-5

Creating source code, 3-4

DELETE command, 2-53, **r23**, 5-31

DIM statement, 2-19, 2-25, **r24**

Dimensioning, 2-26, 2-27, 2-49, **E-4**

differences, 2-26 through 2-28

variables, 2-19, 2-25

E-disk, 1-18

EDIT command, 2-53, **r30**

Editing programs

with IBASIC, 1-9

with the system editor, 1-14, 1-15

ELSE (IF-THEN-ELSE statement), 2-32, **r42**

(EXTENDED IF statement), 2-40, **r36a**

ELSEIF (EXTENDED IF statement), 2-40, **r36a**

ENDIF (EXTENDED IF statement), 2-40, **r36a**

ENDLOOP (LOOP statement), 2-43, **r69a**

ENDSELECT (SELECT statement), 2-45, **r130a**

ENDWHILE (WHILE statement), 2-48, **r170**

END/GO command, 5-8, 5-10

ERR\$ System Variable, 2-19, 2-20

Error messages, 6-1 through 6-17

Runtime and compiler errors, 6-3

XLL error messages, 6-8

XLL I/O error messages, 6-14

XLM error messages, 6-15

Executing the XLL program, 5-4

Executing the XLM program, 5-25

Index

- EXIT command, 5-36
- Exiting the Extended BASIC runtime system, 4-7
- Exiting the Extended BASIC compiler program, 3-16
- EXPORT statement, 2-5, 2-49, r-35a
- Extended BASIC (XBASIC), 2-1
 - command file, C-2
 - compiler program, 3-4, C-1
 - differences from IBASIC, 2-21
 - language, 2-3, 2-4
 - language software, C-1
 - library manager program, 5-3, 5-25, C-1
 - linking loader program, 5-3, 5-4, C-1
 - modified statements, 2-21
 - program development, 2-6
 - runtime system program, 4-3, C-2
 - similarities with IBASIC, 2-4
 - statements, 2-5, 2-34
 - syntax, 2-11
 - unique statements, 2-34, 2-49
- Extended IF statement, 2-40, r36a
- Extended language syntax /E
 - option, 2-12, 3-17
- EXTENDED LIST command, 5-28, 5-32
- Extended syntax, 1-14, 2-11, 2-12
- File names, 2-9, 5-7, 5-27
- File types, new, 3-7
- FIND command, 5-5, 5-11
- FOR-NEXT statements, 2-29, r39
- FORTTRAN interface runtime library, C-2
- FORTTRAN subroutines, E-5
- Getting full use of Extended BASIC, 1-14
- Getting started, 1-3
- Global variables, 2-49, 2-50, 2-51
- GOTO, 2-32, 2-40, r41, r42
- IF-GOTO statement, 2-32, r42
- IF-THEN-ELSE statement, 2-32, r42
- IMPORT statement, 2-51, r42a
- INCLUDE command, 5-5, 5-13
- Integer conversion, 2-18
 - /I option, 3-17
- Integrating subroutines from other languages, E-1
- Interpreted BASIC (IBASIC), 2-3, 2-6
 - compiling an existing program, 3-4
 - unused instructions, 2-52
- Interrupts, 2-30, 2-38
- Labels, 2-5, 2-12, 2-23
 - in branch instructions, 2-5
- Language
 - names, 2-6
 - software, C-1
- LEAVE statement, 2-42, r59a
- Library Manager program, 3-5, 5-3, 5-25
 - commands, 5-28
 - error messages, 5-38, 6-15
 - executing the library manager program, 5-25
 - filename conventions, 5-27
 - terminating the library manager program, 5-25
 - using the library manager program, 5-26
- LINK command, 2-53, r63
- Linkage utility programs, 5-1
- Linking, 1-12, 3-5, 3-22, 3-23
- Linking object files, 3-5
- Linking Loader program, 3-5, 3-23, 5-3, 5-4
 - executing the linking loader program, 5-4
 - error messages, 5-18, 6-8
 - error message table, 5-24
 - commands, 5-8
 - common symbol table, 5-24
 - filename conventions, 5-7
 - map format, 5-19
 - module list format, 5-20
 - symbol table, 5-22
 - terminating the linking loader program, 5-4
 - using the linking loader program, 5-5

- Linking programs, 1-12
- LIST command, 2-53, 5-34, **r64**
- Load map, 5-6, 5-15
- Long variable names in extended syntax, 2-14
- Long variable names in XBASIC, 2-13
- LOOP statement, 2-43, **r69a**

- MAP command, 5-15
- Map format, 5-19
- Memory allocation in Extended BASIC, 3-7
- Memory requirements, 3-6
- MERGE command, 5-35
- Modified statements, 2-21
- Module list, 5-20
- Multidimensional arrays, **E-5**

- New file types, 3-7
- NEXT statement, 2-29
- No line numbers /NL option, 2-17, 3-18
- No markers /NM option, 3-18
- Notation conventions, 2-10

- Object code, 3-4
- Object files, 3-5, 3-7, 3-22
- OLD command, 2-53, **r86**
- Omitted line numbers, 2-17, 3-18
- ON-GOTO statement, 2-30, **r91**
- ON-SUBRET statement, 2-38, **r97a**
- OUTPUT command, 5-17
- Overview of the linkage process, 3-22
 - CTRL /P, 4-7
- Parameter data types, **E-2**
- Parameter passing, **E-2**
- Program development with
 - Extended BASIC, 2-6
- Programs
 - benchmark, 1-9
 - LACE1, 1-15
 - XBCC.COMD, 1-18

- Redimensioning main memory or common arrays, 2-28
- REM statement, 2-31, **r119**
- REPEAT statement, 2-44
- Required programs, **C-1**
 - for linkage, 3-23
- Reserved words, **D-1**
- RESTORE statement, 2-31, **r117**
- RESUME statement, 2-32, **r125**
- Running
 - extended BASIC programs, 3-6, 1-13
 - compiler program, 3-11
 - runtime system program, 4-3
 - runtime system program automatically, 4-3
 - runtime system program from the keyboard, 4-4
- Runtime system, 3-6, 3-22, 4-1
 - error checking, 4-9
 - errors at runtime, 3-6, 4-3, 4-9, 6-3
 - exiting, 4-7
 - loading the program, 4-3
 - messages, 4-8
 - operation, 3-6, 4-3, 4-6
 - program execution, 3-6

- SELECT statement, 2-45, **r130a**
- Software required, 3-23
- Source code, 2-6
- Standard syntax, 2-11
- Statement descriptions, 2-7
- Statement labels, 2-12
- STEP command, 2-52, **r144a**
- STOP statement, 2-31, **r148**
- SUB statement, 2-35, **r149a**
- SUBEND statement, 2-36, **r149b**
- SUBRET statement, 2-37, **r149c**

Index

Subroutines

- array use, 2-25
- names, 2-5, 2-21, 2-22, 2-23
- ON-SUBRET, 2-38, **r97a**
- other languages, E-1
- parameter passing, E-5
- SUB, 2-35, **r149a**
- SUBEND, 2-36, **r149b**
- SUBRET, 2-37, **r149c**
- true, 2-5

Syntax

- differences, 2-11
- extended, 2-11, 2-12 **through** 2-33
- standard, 2-11
- supplementary syntax diagrams, **A-1**

Symbol table, 5-22

System variables, 2-20, **r150**

Tcopy program, 1-4, 1-5

Temporary files, 3-13

Terminating the XLL program, 5-4

Terminating the XLM program, 5-25

Three dimensional arrays, 2-25

TRACE statement, 2-32

True subroutines, 2-5, **see also subroutines**

UNLINK command, 2-53, **r160**

Unique statements in XBASIC, 2-49

UNTIL (REPEAT statement), 2-44, **r123**

Unused interpreted BASIC instructions, 2-52

Using

- Basic compiler program, 3-12
- command files, 1-18, 3-25
- e-disk or a Winchester, 1-18
- runtime system program, 4-6
- XLL program, 5-5
- XLM program, 5-26

Variables

- arrays, 2-19
- differences in, 2-19
- global, 2-49, 2-50, 2-51
- names, 2-22

Virtual arrays, 2-28

WHILE statement, 2-48, **r170**

Winchester disk, 1-18

Writing a program, 1-9