FOR ATARI® COMPUTERS

BASIC XE

A PROGRAMMING LANGUAGE DESIGNED ESPECIALLY FOR THE ATARI 130XE

OSS™
Optimized Systems Software
A Division of ICD, Incorporated
1220 Rock Street, Rockford, IL 61101 (815) 968-2228

A REFERENCE MANUAL FOR

BASIC XE

Optimized Systems Software
A Division of ICD, Inc.

OSS™

1220 Rock Street Rockford, Illinois 61101 (815) 968-2228

Acknowledgements

OSS gratefully thanks Atari, Inc., for its kind permission to reprint portions of the Atari BASIC Reference Manual. Please be aware that these portions have been copyrighted (©) by Atari, Inc., and respect the rights implied thereby.

We also thank those stalwart OSS users whose requests and pleas for an extended BASIC inspired us to create BASIC XE, and those beta-testers who helped us make sure that BASIC XE works the way we want it to.

Trademarks

Preface

You may wonder why BASIC XE needs a reference manual at all. It's just another BASIC, right? Well...yes and no. BASIC XE is another BASIC, but it's a cut above the other BASICs currently available for Atari XL and XE series computers. It needs its own reference manual so that you can find out just how to take advantage of all the extras included in BASIC XE.

What's In This Manual?

This manual does not pretend to teach you how to program in BASIC. There are several very good tutorials that cover the rudiments of BASIC programming on the Atari, and we direct you to them if BASIC is completely foreign to you.

That doesn't mean that this manual is useless. If you want to exploit BASIC XE's advantages, it's a necessity. Between these covers you will find a complete description of the BASIC XE language, including the special statements unique to BASIC XE as well as those in standard BASIC. We have avoided computer jargon whenever possible, resorting to it only when absolutely necessary. To decrease bewilderment we define jargon terms when they are first used, and provide a glossary of all the jargon used in the manual.

As you will notice when you look at the table of contents, this manual groups commands that perform related tasks into chapters, rather than simply listing them in alphabetical order. This enables you to find all the commands that could help you with a specific task. We have included an alphabetized index at the end of the book so that you can find single topics and commands quickly.

Where To Go From Here

If you are planning to read this manual cover to cover before you even boot BASIC XE, that's fantastic! If not, may we suggest that you at least read the introduction and scan the table of contents. This will give you a brief overview of BASIC XE and an idea of where to find things in the reference manual.

Caveat

Because we're only human and so sometimes make mistakes, a caveat is required. We have made every effort to ensure that this manual accurately describes the BASIC XE system and language. However, due to the ongoing improvement and updating of all OSS products (including BASIC XE), we cannot guarantee the absolute accuracy of the documentation. Therefore, OSS, Inc., disclaims all liability for changes, errors, or omissions in either the manual or the software itself.

Extras That BASIC XE Offers You

Of course BASIC XE provides all the commands available in standard Atari BASIC, but that is only the tip of the iceberg. You can LOAD your SAVEd Atari BASIC programs into BASIC XE and make use of its speed immediately, but soon you'll want to take fuller advantage of the extras that BASIC XE offers -- extras like:

Faster Program Execution New floating point math routines combine with the FAST command to produce BASIC programs that execute at near-arcade speed.

Quick Access to the 130XE's Extended Memory Now you can control and utilize the extra 64k of memory in a 130XE, and you don't even have to be a programming genius to do it. One simple BASIC XE statement makes all that space available to your program.

Easy Program Formatting and Editing Unlike other BASICs, BASIC XE does not care whether you use upper or lower case letters when you type in programs. This alone can make your programs more readable. However, BASIC XE will do even more for you. It will automatically prompt you with line numbers or renumber an entire program at your request. Also, the LIST command has a program formatter built in, thus making your programs easier to follow, no matter how complex or involved they are. Other editing features include wrap-around and keyboard repeat. If you enter a program line that's longer than the length of the screen, it will "wrap around" to the next screen line so that you can view it. Also, if you hold down any key for over half a second, it will start repeating.

Advanced String Handling BASIC XE makes string handling easier and more powerful at the same time. No longer must you DIMemsion strings before you use them -- BASIC XE can do it for you. Also, you can now group related strings together in string arrays just like you're used to doing with numbers in numeric arrays. Finally, BASIC XE includes new operators and functions that make string separation, concatenation, and searching a piece of cake.

Built-in Player/Missile Graphics With other BASICs you can use P/M graphics only if you're a computer wiz. BASIC XE provides nine commands designed especially for P/M graphics, and this manual shows you how several others can be applied to P/M graphics. Now P/M graphics are as easy to control as common playfield graphics.

Easier Joystick Control Not only does BASIC XE support the paddle and joystick functions available in Atari BASIC, it also adds several others that make joystick input easier to use.

Explanatory Error Messages Instead of generating a cryptic error number when something goes wrong, BASIC XE also gives you an explanation of the error so that you can diagnose and fix the problem quickly. When you need more help to solve the problem, you can look in Appendix E for a further discussion of error situations.

How to Boot BASIC XE

There's one thing you should do even before you boot BASIC XE for the first time: fill out and return the license agreement that came with BASIC XE. If you don't, you won't be added to OSS's users list, which means that not only will you not get newsletters and update info, but you won't even be able to get technical help from OSS when you call. You must have a license agreement on file to get technical support! So please, please, please, **RETURN YOUR LICENSE AGREEMENT!**

As you have probably noticed by now, BASIC XE is a supercartridge and a disk. To use all of the capabilities of BASIC XE, you need to boot with both the cart. and the disk. The process is simple:

1) Turn on drive 1, making sure that it's connected to your computer.
2) Insert the BASIC XE Extensions Disk in drive 1 and close the drive door.
3) Insert the BASIC XE cartridge in your computer.
4) Turn on your computer and wait.

Soon you will see a title screen telling you that the extensions are loading. After this the screen will clear and you will see the BASIC XE copyright message at the top of the screen, and the familiar Ready prompt will appear right below that. Now you're ready to program!

You can boot without the extensions disk if you want. One of two things will happen, depending upon whether the disk you boot with has the extensions file on it (instructions for copying the extensions disk and file are below).

If the boot disk does not have the extensions file on it, or if you boot without a drive, you can still use BASIC XE. However, the following will not be available:

BSAVE, CALL, DEL, EXIT, FAST, LOCAL, LVAR, MOVE, PROCEDURE, RENUM, RGET, RPUT, SORTUP, SORTDOWN, the fast math routines, and all P/M commands except **HITCLR.**

If the boot disk does have the extensions file on it, you will be able to use all of the capabilities of BASIC XE, just as if you had booted with the extensions disk.

Backing Up the Extensions Disk

The extensions disk is in single density Atari DOS 2.0s format, so duplicate it using whatever command your DOS requires to duplicate this disk format.

Moving the Extensions to Other DOS's

The BASIC XE extensions are in the file BASICXE.OSS on the extensions disk. If you want to use a DOS other than the one on the extensions disk, all you have to do is copy the BASICXE.OSS file to your DOS boot diskette. This file is in standard DOS LOAD format, so copying it should not be a problem.

Warning: BASIC XE will not work with any 'translator' program, nor will it work with DOSXL.SUP or OurDOS if you use the extensions (because they try to use the same memory).

How to Use this Manual

This section might seem superfluous because everybody knows how to use a manual. That may be true, but all manuals have their own idiosyncracies, even this one, and we thought you might want to know them.

The chapter groupings were designed around topics so that you can find out everything about a single topic without having to jump from place to place. Also, the chapters themselves have been grouped into larger topical groups (e.g., the Graphics and P/M Graphics chapters are together), with the simpler topics near the beginning of the book. If you are looking for something specific, use the index. it contains a multitude of references, including subheadings within larger entries. Finally, if a topic confuses you, try the examples. That's what they're there for!

Special Notations this Manual Uses

This manual's job is to teach you how to use BASIC XE and its extensions without befuddling you. To this end we have adopted several conventions in our presentation of the language. We list them here at the beginning so that you can familiarize yourself with them:

Capitalized Words In the text of this manual, all keywords and functions are printed in uppercase to differentiate them from the other parts of a statement.

Lowercase Words In the text of this manual, lowercase words are used to denote the various classes of items which may be used in a program, such as variables (var), expressions (exp), etc.

Abbreviations in Section Headings If a statement has an abbreviation associated with it, the abbreviation is placed in parentheses following the full name of the statement in the heading (e.g., LIST (L.)).

An "f" Preceding a Keyword If an "f" precedes a Keyword in a section heading, it means that the Keyword is a function, not a statement.

Items in Brackets When showing the usage format of statements and functions, we use brackets ([]) to surround items which are optional in the format. If the item enclosed in brackets is followed by an ellipsis (three dots), it means that item may be used zero or more times in the format (e.g., [exp,...] means that you may use 0,1,2,3, or more expressions, separated by commas).

Items Stacked in Bars Items stacked vertically in bars indicate that any one of the stacked items may be used, but that only one at a time is permissible. In the following example, you may either use the GOTO or the GOSUB, but not both:

```
|GOTO  | 2000
|GOSUB|
```

Notes, Cautions, and Warnings: You will find these starting paragraphs throughout this manual. Notes are simply interesting asides, Cautions are just that (they point out things to watch out for), and Warnings describe potentially catastrophic situations and problems.

BASIC XE's Operating Modes

We humans don't like to do things the same way every time, but computers do. BASIC XE solves this problem by having three "operating modes". This helps keep you and BASIC XE working on the same wavelength. The following paragraphs describe these modes and outline what each is used for.

Direct Mode This is the mode you're in whenever you see the "Ready" (or "XE Ready" if you've used the **EXTEND** statement) prompt. For this reason Direct Mode is sometimes called Prompt Mode. Commands you issue in this mode are executed immediately (Directly). Most of the time you will use this mode only to tell BASIC XE what you want to do next.

Deferred Mode You enter this mode when you use the **NUM** command, type in a line that begins with a line number, or edit a program line. Commands you issue in this mode will not be executed until you tell BASIC XE to do so. For this reason Deferred Mode is sometimes called Program Mode. When you tell BASIC XE to execute a program (i.e., some numbered lines), it will use the line numbers to determine the order in which you want the program executed.

Execute Mode BASIC XE goes into this mode when you tell it to start executing a program and will remain in it until the program halts. The halt can occur before the program is finished if the program causes an error, or if you press **BREAK** or **SYSTEM RESET**.

BASIC XE Keywords and Symbols

The following table shows all the words and symbols that mean something special to BASIC XE:

ABS	DATA	FRE	LVAR	PMWIDTH	RUN	TRAP	
ADR	DEG	GET	MID$	POINT	SAVE	UNPROTECT	
AND	DEL	GOSUB	MISSILE	POKE	SET	USING	
ASC	DIM	GOTO	MOVE	POP	SETCOLOR	USR	
ATN	DIR	GRAPHICS	NEW	POSITION	SGN	VAL	
BGET	DOS	HEX$	NEXT	POINT	SIN	VSTICK	
BLOAD	DPEEK	HITCLR	NORMAL	PROCEDURE	SORTDOWN	WHILE	
BPUT	DPOKE	HSTICK	NOT	PROTECT	SORTUP	XIO	
BSAVE	DRAWTO	IF	NOTE	PTRIG	SOUND	!	"
BUMP	ELSE	INPUT	NUM	PUT	SQR	#	$
BYE	END	INT	ON	RAD	STATUS	%	&
CALL	ENDIF	INVERSE	OPEN	RANDOM	STEP	(	)
CHR	ENDWHILE	LEFT$	OR	READ	STICK	*	/
CLOAD	ENTER	LEN	PADDLE	REM	STOP	+	-
CLOG	ERASE	LET	PEEK	RENAME	STR$	,	<
CLOSE	ERR	LIST	PEN	RENUM	STRIG	<=	<>
CLR	EXIT	LOAD	PLOT	RESTORE	SYS	=	>
COLOR	EXP	LOCAL	PMADR	RETURN	TAB	>=	^
CONT	EXTEND	LOCATE	PMCLR	RGET	THEN	;	:
COS	FAST	LOG	PMCOLOR	RIGHT$	TO		
CP	FIND	LOMEM	PMGRAPHICS	RND	TRACE		
CSAVE	FOR	LPRINT	PMMOVE	RPUT	TRACEOFF		

A Glossary of Terms this Manual Uses

adata Short for "ATASCII Data". Any ATASCII character, excluding commas and carriage returns. (see DATA for more info.)

aexp Short for "arithmetic expression".

alphanumeric The letters A through Z (either lower or upper case) and the digits 0 through 9.

aop Short for "arithmetic operator".

Arithmetic Expression An expression that evaluates to a number. For more information, see the **Expressions** chapter.

Arithmetic Operator A unary or binary operator that performs a math operation.

Arithmetic Variable A location where a single number is stored.

Array A one-dimensional structure in which each element (cell) is uniquely described by its element number. The **Variables** chapter gives a more in-depth definition.

avar Short for "Arithmetic Variable".

Binary Anything that has two states (on/off, up/down, action/stasis, etc.) Not simply "a number system based on powers of 2".

Channel See the Introducing Atari I/O section of the **Beginning Data Input/Output** chapter for a complete discussion.

cname Short for "Calling Name". The name used to CALL a **PROCEDURE**; may be either a string constant or svar. **Note:** substrings and savars may not be used.

Command Anything you tell BASIC XE to do is a command, so both statements and functions are commands. If you give a command in Direct Mode it will be executed immediately, but if you're in Deferred Mode BASIC XE will not execute the command until you tell it to do so.

Device A peripheral (add-on) that you can use for I/O. The Introducing Atari I/O section of the **Beginning Data Input/Output** chapter discusses this term in further detail.

exp Short for "expression".

Expression An expression is any legal combination of variables, constants, operators, and functions used together to compute a value. Expressions can be either arithmetic or string.

Floating Point Numbers represented using a decimal point (4.5, -28.49)

filespec Short for "file specifier". A filespec is used when when doing some types of I/O. You can find a complete definition of this term in the Introducing Atari I/O section of the **Beginning Data Input/Output** chapter.

Function A function is a subroutine built into the computer so that it can be called by your program. Functions and statements differ in that functions must be used in expressions to accomplish their task, whereas statements are selfsufficient. COS (Cosine), FRE (remaining memory), and INT (integer) are examples of functions.

Integer A whole number (not a fraction). Integers may be either positive (4, 183) or negative (-4, -183).

I/O Short for "Input or Output". This term refers to the transfer of data between your computer or BASIC program and peripheral devices like printers, disk drives, etc.

Keyword Any word that means something special in the BASIC XE language.

lineno Short for "line number". A constant that identifies a particular program line. Must be an integer from 0 through 32767. Line numbering determines the order of program execution.

Literal String A synonym of "String Constant".

Logical Operator An operator that performs a comparision where the result is either "true" (1) or "false" (0).

lop Short for "Logical Operator".

Matrix A two-dimensional structure composed of separate elements. Each element (cell) in a matrix is uniquely described by its row and column number.

Matrix Variable An arithmetic variable of 1 (an array) or 2 (matrix) dimensions. See the mvar section of the **Variables** chapter for more info.

mvar Short for "matrix variable".

Numeric A synonym of "Arithmetic".

Operator Operators are used in expressions to tell BASIC XE how it should evaluate the variables, constants, and functions in the expression. There are two operator types: arithmetic and logical.

pexp Short for "Passing Expression". An expression whose value will be passed passed via **CALL** to a **PROCEDURE**, or passed via **EXIT** back to the **CALL**. **pexp** may be an exp, avar, svar, savar, or mvar. **Note:** svars, savars, and mvars must be preceded by a !.

pmnum A player or missile number in P/M Graphics. Players are numbered 0-3, and missiles 4-7.

pname Short for "Procedure Name". The name used to identify a **PROCEDURE**. pname must be a string constant.

Program Line BASIC XE program lines are made up of three elements: the line number, the program statement(s) (multiple statements are separated by colons), and the line terminator (a **RETURN**). In an actual program, the three elements might look like this:
100 PRINT "I'm a program line.":GOTO 100
If a program line will not fit on one screen line, it will wrap around to the next screen line so that you can see the entire program line.

rvar Short for "Receiving Variable". A var which will receive a the value of a parameter passed either from **CALL** to **PROCEDURE**, or from **EXIT** back to **CALL**. **Note:** svars, savars, and mvars must be preceded by a !.

savar Short for "String Array Variable".

sexp Short for "String Expression".

Statement Statements are subroutines built into BASIC XE that will perform specific tasks for you. Statements and functions differ in that functions must be used in expressions to accomplish their task, whereas statements are selfsufficient.

String Constant A group of characters enclosed in quotation marks. "OSS is the best" is a string constant. So are "123456789" and "Hello".

String Expression An expression that evaluates to a string constant. May consist of an svar, an savar element, a string constant, or a function that returns a string constant.

String Variable A variable where a single string is stored.

String Array Variable An array variable whose elements are strings.

Substring Simply a part of a string (e.g., "abc" is a substring of "abcdef").

svar Short for "String Variable.

var Short for "Variable".

Variable This is the term used to describe a quantity which may (or may not) change. In BASIC XE, there are two basic types of variables: string and arithmetic.

Your Additions to the Glossary

Types of Variables

BASIC XE supports two basic types of variables: arithmetic variables and string variables. In addition, it supports both arithmetic and strings arrays, and arithmetic matrices. Arithmetic variables, arrays, and matrices are used to store numbers, and may be used only where numbers are required. String variables and arrays store character strings and may be used only where a character string is required.

Variable Names

All variable names must start with an alphabetic letter, but the rest of the characters in the name may be either letters or digits. Also, variable names must be less than 120 characters long. Finally, string varable and array names must end with the dollar sign ($) character. The following examples should make these requirements clearer:

Arithmetic Names	String Names
Rate	Name$
Player1score	A$
Temp	Title$

Number of Variables

BASIC XE limits you to a maximum of 128 variables. If you need more than 128 (which is unlikely), you might use elements of an array as individual variables instead of having a separate name for each. You might also use **LOCAL** to create reusable private variables. To clear the variable name table of extraneous names (possibly after an error 4), **LIST** your program to disk or cassette, type **NEW** to clear the variable name table, and then **ENTER** your program back into memory. We suggest that you use **SET 5,0** and **SET 12,0** before doing this.

Arithmetic Variables (avar)

Arithmetic variables are used to store numbers, and are the most common variables used. Here are some examples of arithmetic variables in use:

```
100 Input "avar Value)> ",X
110 Print "X:       ";X
120 Print "X^2:     ";X^2
130 Print "√X:      ";X^0.5
140 Print "e^X:     ";Exp(X)
150 Print "ln(X):   ";Log(X)
160 Print "log(X):  ";Clog(X)
170 Print :Goto 100
```

Arithmetic Arrays and Matrices (mvar)

An arithmetic array is a group of separate arithmetic variables (called elements or subscripts of the array) which share a common name, and may accessed only by specifying the number of a given element as well as the name of the arithmetic array. If you think of an array as a string of pearls the idea is easier to understand. If you want to list the worth of each pearl (for insurance purposes), your list might look like:

```
Pearl 1: $1000.00
Pearl 2:  $950.00
Pearl 3: $1125.00
Pearl 4: $1100.00
Pearl 5: $1050.00
Pearl 6: $1200.00
```

Translated into a BASIC XE arithmetic array, your list would be:

```
100 Dim Pearl(5)
110 Pearl(0)=1000
120 Pearl(1)=950
130 Pearl(2)=1125
140 Pearl(3)=1100
150 Pearl(4)=1050
160 Pearl(5)=1200
```

Notice that the elements of the BASIC XE arithmetic array are numbered starting at zero. This doesn't seem right because we humans don't think of zero as a number, but - as far as computers and mathematicians are concerned - it is.

The **DIM** statement on line 100 is used to tell BASIC XE how many elements you want reserved for the arithmetic array named "Pearl". **DIM** is discussed in greater detail in its own section later in this chapter.

An arithmetic matrix is similar to an arithmetic array, except that it is two dimensional. This means that there are two numbers required to specify a given element: a row number and a column number. Our string of pearls analogy can be extended to describe matrices if you consider a matrix as a bunch of pearl strings. Now, your price list would look something like:

String 1	String 2	String 3
Pearl 1: $1000.00	Pearl 1: $875.00	Pearl 1: $1100.00
Pearl 2: $950.00	Pearl 2: $1075.00	Pearl 2: $980.00
Pearl 3: $1125.00	Pearl 3: $1300.00	Pearl 3: $1115.00
Pearl 4: $1100.00	Pearl 4: $990.00	Pearl 4: $1120.00
Pearl 5: $1050.00	Pearl 5: $1250.00	Pearl 5: $890.00
Pearl 6: $1200.00	Pearl 6: $1035.00	Pearl 6: $1225.00

Translated into a BASIC XE arithmetic matrix, your list would be:

```
100 Dim Pearls(2,5)
110 Pearls(0,0)=1000:Pearls(1,0)=875:Pearls(2,0)=1100
120 Pearls(0,1)=950:Pearls(1,1)=1075:Pearls(2,1)=980
130 Pearls(0,2)=1125:Pearls(1,2)=1300:Pearls(2,2)=1115
140 Pearls(0,3)=1100:Pearls(1,3)=990:Pearls(2,3)=1120
150 Pearls(0,4)=1050:Pearls(1,4)=1250:Pearls(2,4)=890
160 Pearls(0,5)=1200:Pearls(1,5)=1035:Pearls(2,5)=1225
```

As with arithmetic arrays, the first element index is 0 rather than 1, so the first pearl on the first string is accessed using the subscript (0,0). The first 0 is the number of the pearl string (the row number), and the second is the number of the individual pearl (the column number). This analogy might lead you to believe that a matrix is just an array where each element is itself an array (our list is one of strings of pearls, and each string of pearls is a group of individual pearls). This conception of matrices is, in essence, correct and is very useful when trying to manipulate matrices.

When you use a single element of an arithmetic array or matrix, you are actually using a single number (which is what an arithmetic variable is). This means that **avar**, **array(element)**, and **matrix(row,column)** may all be used whenever a number is wanted.

String Variables (svar)

String variables are used to store literal strings of characters. A literal string of characters is simply some characters enclosed in double quotes; for example,

```
"This string enclosed in quotes is a literal string"
"Numbers in quotes are strings too - 12345"
"Even control charcters are - ├╠┤┐◢◣▲"
```

are all literal strings. As mentioned earlier, string variable names are just like arithmetic variable names, except that they must end with a dollar sign ($).

Before you use a string variable, you need to tell BASIC XE the size (maximum number of characters) of the variable. This is done using the **DIM** (dimension) statement as follows:

```
DIM String$(66), A$(10)
```

Note: When you manipulate strings a character at a time, remember that the element numbering begins at 1, not 0 (as with arithmetic arrays and matrices). For example, if you want to get the first character of A$ (which contains the string "ABCDEFG"), you would use A$(1,1), and get "A" as the result. If you try to get the "A" by using A$(0,0), you will get an error.

Bonus: BASIC XE can automatically dimension a string variable for you if you don't manually **DIM**ension it. For more information about this feature see the discussion of **SET 11,aexp**.

String Array Variables (savar)

A string array is very similar to an arithmetic array, except that each element is a string variable, not an arithmetic variable.

String array variables resemble string variables in three aspects: their names must end with a dollar sign, they must be **DIM**ensioned before being used, and their element numbering begins at 1, not 0. However, there are two dimensions to a string array: the number of strings in the array, and the length of the strings. The following examples show how to specify both of these dimensions:

```
DIM Sarray$(4,40), A$(10,100)
```

This example first dimensions a string array called "Sarray$" to contain 4 strings, each 40 characters long, and then dimensions "A$" to 10 strings, each 100 characters long.

To access one of the strings in a string array you specify the string's number (remember, the first string is number 1, not 0) followed by a semicolon (;), as follows:

```
100 Dim Test$(3,5)
110 Test$(1;)="This "
120 Test$(2;)="is a "
130 Test$(3;)="test."
```

As you may notice, savar(element;) is equivalent to svar, and may be used wherever svar is used, unless stated otherwise.

DIM

Format: DIM | mvar(aexp1[,aexp2]) | svar(aexp1) | savar(aexp1,aexp2) | [,...]

The **DIM** statement is used to reserve space for arithmetic arrays and matrices, and strings and string arrays.

For arithmetic arrays **DIM** reserves space for **aexp1+1** arithmetic elements. For arithmetic matrices it reserves space for **aexp1+1** rows of **aexp2+1** elements each. The "+1" is there because arithmetic indexing begins at 0, thus giving you aexp+1 total indices.

DIM reserves space for up to **aexp1** characters when allocating strings, and space for **aexp1** strings, each of up to **aexp2** characters, when allocating string arrays.

The following examples illustrate the use and effect of the **DIM** statement. The first one reserves 101 arithmetic elements for an array named **A1**. The second allocates space for 7 rows of 4 columns each for a matrix called **Grid**. The last example reserves 20 bytes for the string **Bstr$**, and then allocates 100 strings, each of up to 40 characters, for the string array **Friends$**.

```
100 Dim A1(100)
110 Dim Grid(6,3)
120 Dim Bstr$(20),Friends$(100,40)
```

Note: BASIC XE is capable of automatically DIMensioning string variables. For more information, see **SET 11,aexp**.

LOCAL

Format: LOCAL avar1 [,avar2...]

Examples: 100 LOCAL Temp1
320 LOCAL Sum,N,Count,Misc

The **LOCAL** statement allows you more flexibility in your programming because it enables you to have temporary arithmetic variables within **PROCEDURE** and **GOSUB** subroutines. The way **LOCAL** works is very simple. When a **LOCAL** statement is executed, all avar names (no mvars, svars, or savars) following it become private until the next **EXIT** is encountered. What does 'become private' mean? Simply that you can change the value of a **LOCAL** avar within its **LOCAL/EXIT** bounds without affecting its value outside of these bounds, as if you had a private copy of the variable. When you use **LOCAL**, you don't have to worry about conflicts between routines in your program that use variables with the same name.

A simple example will help:

```
10 Test=1234567:Print 10,Test
20 Gosub 40:Print 20,Test
30 End
40 Local Test:Print 40,Test
50 Test=0.54321:Print 50,Test
60 Exit
```

Note the that **PRINT** statements purposely display the current line number as well as the value of **Test**. This is simply to make tracing the flow of the program easier. Does it surprise you to find that the output of the above program will look something like this?

```
10      1234567
40      1234567
50      0.54321
20      1234567
```

Let's examine that program a little closer. Line 10 is simple enough - we just assign a value to the variable **Test** and verify that it has been accepted. In line 20, we first **GOSUB** to a routine and then again display the contents of our variable. Note that in the program's running this **PRINT** is the last thing executed (other than the **END**). Line 40 begins the interesting part of this program. We declare that **Test** is a **LOCAL** variable and, once again, display its value. Line 50 is a repeat of line 10 except that we assign a different value to our now-private variable **Test**. Note that the **PRINT** verifies our change. Finally, in line 60, we use **EXIT** to restore **Test** to its original value, as shown by the **PRINT** in line 20.

The point of all this was to show that our subroutine (lines 40 through 60) could do what it liked with the **LOCAL** variable without affecting its value in the rest of the program.

Bonus: when you **POP** a **LOCAL** variable the non-private value is restored, so you can use **LOCAL** and **POP** to create private variables even when you're not in a subroutine.

Notes and Warnings Regarding LOCAL

Note: the fact that **LOCAL** may be used with **GOSUB** subroutines is not an accident. **EXIT** was specially designed to find out what type of subroutine (**PROCEDURE** or **GOSUB**) it is terminating, and handle the returning condition appropriately. This small fact alone allows you to modify your existing programs to use **LOCAL** variables without having to change all **GOSUB**s to **CALL**s. Also, there are occasions where it could be advantageous to use **GOSUB** instead of **CALL**. In particular, **GOSUB**bing to an absolute line number is significantly quicker than any other type of subroutine access when your program is in **FAST** mode.

Note: variables do not change value when they are made **LOCAL**. You can see this in the example earlier in this section. The **PRINT**ed value of Test in line 40 is still 1234567, even though it has been made private. If you want your **LOCAL** variables to be zeroed before you use them, you must equate them to zero yourself.

Note: since you are still limited to 128 different variable names, you might consider using the same **LOCAL** variable names in all your subroutines if you are pushing the name limit. For example, you might start each subroutine with the line

Each subroutine then has four variables available exclusively for its own use, and you have used only four names from your maximum of 128.

Technical Note: **LOCAL** pushes the current value of an avar onto BASIC XE's stack when that variable is made private. When an **EXIT** is encountered, the value is popped off the stack and into the avar, thus restoring its previous value.

Warning: you may use **LOCAL** only at the beginning of subroutines that are terminated by an **EXIT** (not a **RETURN**), unless you **POP** the previous values before **RETURN**ing. For more info, see **POP**.

Assigning Values to Variables

The assignment statement is used to assign a value to a variable, and is of the general form variable=expression. The variable and expression must be of the same data type (arithmetic or string) or you will get an error.

Arithmetic Assignment

Arithmetic assignment is the simplier of the two, so we'll discuss it first. The syntax is simple: avar=aexp, but the extensions are numerous. When you remember that subscripted arithmetic arrays and matrices are functionally equivalent to simple arithmetic variables, all of the following become valid:

```
100 Dim Array(10),Matrix(10,10)
120 Arithvar=27.4
130 Matrix(0,0)=27.4
```

String Assignment

String assignment can be done two ways: by substring and by entire string. Before discussing these two methods, we need to discuss what "string" and "substring" mean. The following table defines these terms when used as both as the source and destination in an operation (e.g., in A$="abc", A$ is the destination, and "abc" is the source):

String	As Source String	As Destination String
S$	characters 1..LEN value	characters 1..DIM value
S$(n)	characters n..LEN value	characters n..DIM value
S$(n,m)	characters n..m	characters n..m

Assigning an entire string is easy; the form is simply svar=sexp. Whatever svar had in it before is wiped out and sexp is put in. The LEN value is set to the length of the sexp string. Here are some examples:

```
10 Dim S1$(50),S2$(50)
20 S1$="A string assignment"
30 S2$="Another string assignment"
```

Substring assignment can be done using either the format svar(n,m)=sexp or svar(n)=sexp. In the first case, characters n through m (inclusive) of svar will be changed to sexp. If sexp evaluates to a string longer than the specified destination substring, only the characters up to the substring length will be assigned. If the sexp string has fewer characters than the destination substring, only LEN(sexp) characters will be changed in the substring. Also, BASIC XE will update the length of svar if the substring assignment makes it longer. The second method of substring assignment replaces n through the DIM value of svar with the sexp string, and then updates the length of svar. The example on line 90 illustrates this type of substring assignment. The others show the two subscript method:

```
40 Rem "Use DIM's from above"
50 S1$="ABCD"
60 S1$(4,8)="1234":Rem S1$="ABCD1234"
70 S1$(1,4)="ab":Rem S1$="abCD1234"
80 S2$="BASIC XE - Precision Software"
90 S2$(10)="FROM OSS":Rem "S2$=BASIC XE from OSS"
```

To assign a value to a string array (savar), first you specify which string element of the savar you want to use (followed by a semi-colon), and then treat it just like a normal string (svar). The following examples help clarify this procedure:

```
10 Dim Sa$(10,40)
20 Sa$(1;)="A string assignment":Rem "savar version of 20 above"
30 Sa$(2;)="ABCD"
40 Sa$(2;4,0)="123456":Rem "savar version of 60 above"
50 Sa$(3;)="BASIC XE - Precision Software"
60 Sa$(3;10)="from 055":Rem "savar version of 90 above"
```

BASIC XE also allows you to do string concatenation (tacking one string onto the end of another) easily using the assignment statement. To concatenate strings, simply change the **sexp** in the string assignment format to **sexp1,sexp2,sexp3,....** **sexp2** is then concatenated to **sexp1**, **sexp3** is concatenated to the result, and so on. The following examples show concatenation:

```
10 Dim A$(10),B$(20),C$(40)
20 A$=" from 055"
30 B$="BASIC XE"
40 C$=B$," a hot language",A$
50 B$=B$,A$
60 Print C$:Print B$
```

Note that line 50 is equivalent to

```
50 B$(Len(B$)+1)=A$
```

Note: it is possible to store into the middle of a string by using subscripting; however, the beginning of the string will contain garbage or nulls.

LET

Format: LET <assignment statement>
Example: LET GOTO=3.5
LET LETTERS$="a"

LET allows you to assign values to variables with names that start with or are identical to a keyword. In the first example, **LET** allows **GOTO** to be used as an arithmetic variable rather than as the **GOTO** statement. The second allows the use of **LETTERS$**, the first the letters of which are the keyword **LET**.

There are a few keywords which CANNOT be used as variable names even when you use **LET**. They are the unary logical operator **NOT**, and all the function names (**ABS**, **LEN**, etc.) Here is an example of what will happen if you try to use **NOT** as the first three letters of a name. Type in this program:

```
10 CSHARP=37
20 LET NOTE=CSHARP
30 PRINT NOTE
```

When you **RUN** it, a "1" will get printed on the screen, not a "37". If you **LIST** the program you will see why. Line 30 is listed as

```
30 Print  Not E
```

because BASIC XE does not allow "NOT" as the start of a variable name and interprets it as the keyword **NOT**.

Space For Your Notes

Operators

BASIC XE has two types of operators: Arithmetic Operators and Logical Operators. As you will see in the expressions chapter, either of these two types of operators may be used in arithmetic expressions, while neither may be used in string expressions.

Before discussing these two types of operators, a reminder of the meaning of 'binary' is needed. As stated in the glossary, this term does not mean simply "a number system based on powers of 2, in which 0 and 1 are the only digits". When 'binary' is used to mean this, it is an abbreviation of 'binary number system', and applies only to numeric representations within this system. Anything which has only two states (on and off, up and down, action and stasis, etc.) can be considered binary. When we are discussing operators, 'binary' means that the operator requires two operands. For example, * is a binary operator because it multiplies one value by a second (4*3 means something, while *3 means nothing). Similarly, 'unary' is used to describe an operator which requires one operand (- is a unary operator when we use it to signify that a number is negative, e.g. -5).

Arithmetic Operators (aop)

BASIC XE supports 8 binary and 2 unary arithmetic operators. The binary ones are:

Symbol	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
&	Bitwise AND
!	Bitwise OR
%	Bitwise EOR (Exclusive OR)

The first four are straightforward enough since they are the arithmetic operators we use all the time, but the last four require some explanation.

The ^ operator is used to raise a number to a specified power. For example, 4^3 simply means "multiply 4 by itself 3 times", or 4*4*4, which equals 64.

The &, !, and % operators allow you to perform bitwise operations on positive integers up to 65,535. If you use them with non-integers (e.g., 4.3, 9.528, etc.), the number will be rounded to the nearest integer before the operation. If you try to use them with negative numbers an error occurs. The following tables show the results of comparing two bits for each of these operators:

Bit A	Bit B	Result
1 &	1 =	1
0 &	1 =	0
1 &	0 =	0
0 &	0 =	0

Bit A	Bit B	Result
1 !	1 =	1
0 !	1 =	1
1 !	0 =	1
0 !	0 =	0

Bit A	Bit B	Result
1 %	1 =	0
0 %	1 =	1
1 %	0 =	1
0 %	0 =	0

The following examples illustrate the results of using each of these bitwise operators with the operands 5 and 39:

```
& example          ! example          % example
  00000101 (5)       00000101 (5)       00000101 (5)
& 00100111 (39)    ! 00100111 (39)    % 00100111 (39)
  00000101 (5)       00100111 (39)      00100010 (32)
```

The two unary arithmetic operators are plus (+) and minus (-), and are used to denote the sign (positive/negative) of a number. For example, +5 means "positive five" and -5 means "negative five". **Note:** If you do not specify the sign of a number, BASIC XE assumes that the number is positive.

Logical Operators (lop)

BASIC XE supports three types of logical operators: relational, unary and binary.

The relational operators compare two expressions, giving a boolean (true/false) result, and are most frequently used in conditional statements (i.e., the IF statements). They may also be used in arithmetic expressions, returning a 1 if the relation is true, and a 0 if it's false.

```
<   The first exp is less than the second exp.
>   The first exp is greater than the second.
=   The exps are equal to each other.
<=  The first exp is less than or equal to the second.
>=  The first exp is greater than or equal to the second.
<>  The two exps are not equal to each other.
```

Examples of the relational lops may be found in the **Expressions** chapter.

The unary logical operator is **NOT**, and is used to reverse the result of an expression. For example, the expression **2<3** is obviously true, but the expression **NOT(2<3)** is false, since NOT inverts the truth of "2 is less than 3".

There are two binary logical operators: **AND** and **OR**. Do not confuse them with the bitwise binary arithmetic operators & and !. They are not the same! **AND** and **OR** are used to create compound logical expressions like

```
IF X=3 OR Y=9 THEN GOTO 400
WHILE Done=0 AND Bail=0
```

Note how these operators are different. Only one of the two operand expressions must be true for the logical **OR** to be true, while both must be true for the logical **AND** to be true.

Operator Precedence

Operators require some kind of precedence (a defined order of evaluation) or we wouldn't know how to evaluate expressions like **4+5*3**. Is this equal to (4+5)*3 or 4+(5*3)? Without operator precedence it's impossible to tell. BASIC XE's normal precedence is very precise, as shown in the following table. The operators are listed in order of highest to lowest precedence. Operators on the same line are evaluated left to right in an expression.

( )	Parentheses
< > = <= >= <>	Rel. lops in String Comparisions
NOT + -	Unary NOT lop, Unary Plus and Minus aops
^	Exponentiation
% ! &	Bitwise EOR, OR, AND aops
*** /**	Binary Multiplicative aops
+ -	Binary Additive aops
< > = <= >= <>	Rel. lops in Numeric Comparisons
AND	Binary AND lop
OR	Binary OR lop

If you're ever in a situation where you're unsure of the evaluation of an expression, use parentheses to insure the proper order of evaluation. Examples of operator precedence during expression evaluation can be found in the **Expressions** chapter.

Space For Your Notes

Expressions

Expressions are constructions which obtain values from variables, constants, and functions using a specific set of operators. BASIC XE supports two types of expressions: arithmetic (aexp) and string (sexp). Before discussing these two types of expressions something needs to be said about the constants BASIC XE allows.

String and Numeric Constants

String constants are frequently called literal strings because they are just a group of characters enclosed in double quotes ("):

```
"This string enclosed in quotes is a string constant"
"Numbers in quotes are strings too - 12345"
"So are control charcters are - ├┼┤┬┴┘"
```

To get a double quote into a string constant, use two double quotes in a row ("").

BASIC XE allows you to enter numeric constants (numbers) in one of two ways - decimal or hexadecimal. Decimal numbers may either be integers, fractions, or scientific notation. The following examples illustrate these three types of numbers:

Integers	Fractions	Sci. Notation
4027	-67.254	4.33E2
-2	325.04	23.4E-14

The "E" in the scientific notation examples stands for "Exponent". The number following it is the power of ten (e.g., 4.33E2 means "4.33 * 10^{2}", or 433).

Hexadecimal numbers can only be integers, and the digits must be preceded by a dollar sign ($), as in the following examples:

```
$4A30     -$0A    $6FF
  -$E   -$7B2D   $FFFF
```

Notice that the unary minus (denoting a negative number) precedes the dollar sign. The maximum hexadecimal value allowed is $FFFF (65,535 decimal).

Internal Format of Numbers

Note: this section is provided for those of you who are interested in the technical aspects of BASIC XE. You can skip this section without impairing your ability to use BASIC XE.

All numbers in BASIC XE are Binary Coded Decimal (BCD) floating pointing point with a five byte (10 BCD digit) mantissa and a one byte exponent. The most significant bit of the exponent is the sign of the mantissa (0 for positive, 1 for negative), and the rest of the bits are the value of the exponent in excess 64 notation. Internally, the exponent represents powers of 100 (not powers of 10). For example, 0.02 equals $2*10^{-2}$, which equals $2*100^{-1}$, so the internal representation is

```
3F 02 00 00 00 00
```

$3F is the exponent (-1) plus 64 ($40), and the mantissa is 2. The implied decimal point is always to the right of the first byte of the mantissa. An exponent less

than $40 indicates a number between 0 and 1, while an exponent greater than or equal to $40 represents a number greater than or equal to 1. Zero is represented by a zero mantissa and a zero exponent.

In general, numbers have a 9 digit precision. For example, only the first 9 digits are guaranteed to be significant when INPUTting a number. You can sometimes get 10 significant digits in the special case where an even number of digits are to the right of the decimal point.

Arithmetic Expressions (aexp)

Arithmetic expressions are those which evaluate to a number, and are made up of one or more of the following list of operands, separated by operators:

1) a numeric constant (number)
2) an avar (or subscripted mvar)
3) a function which returns a number
4) string comparision using relational lops

The first three are straightforward, but the fourth requires explanation. You may use string comparisions in arithmetic expressions because the comparision results in a 1 (true) or 0 (false). For example, "ABC"<"ACC" would return a 1, since "ABC" precedes "ACC" when the two are alphabetized. Conversely, "ABC">"ACC" evaluates to 0. An arithmetic expression can simply be one of the above described operands, or two or more of them separated by operators (either arithmetic or logical). The following examples of arithmetic expressions include the evaluation order of the operators (if any) and the result:

Expression	Evaluation Order	Result
3*(4+(21/7)*2)	/,*,+,*	30
"AB">"AC"+7*(ASC("A"))	>,ASC,*,+	455
X=100 : Y=2	N/A	
INT(X*Y/3)	*,/,INT	66

String Expressions (sexp)

String expressions are much simpler than arithmetic expressions since there are fewer things they can be. The following list shows all the valid string expressions:

1) a string constant (literal string)
2) an svar (or subscripted savar)
3) a function which returns a string
4) a substring of an svar or savar

Notice that nothing has been said about operators in string expressions. That is because none are allowed (with the special exception of the comma (,) for concatenation in string assignment). A string expression may be only one of the above, as in the following examples:

```
"A literal string"      A$(3)
A$                      Sa$(1;3)
Sa$(1;)                 A$(4,8)
STR$(126.83)            Sa$(1;4,8)
```

Editing Your Program

The statements in this chapter ease the job of editing a BASIC XE program, so that programming need not be considered a chore. This chapter covers the statements **NEW**, **NUM**, **LIST**, **DEL**, **RENUM**, and **REM**.

NEW

Format: NEW
Examples: NEW
100 NEW

This command erases the BASIC XE program currently in memory. Therefore, before typing **NEW**, make sure you have saved your program (using SAVE, CSAVE or LIST) if you want to keep it. **NEW** also clears BASIC XE's internal symbol table so that no variables are defined. **NEW** is normally used in Direct Mode but is sometimes useful in Deferred Mode as an alternative to **END**, when you want a program wiped out after it has **RUN**.

NUM

Format: NUM [start][,inc]
Examples: NUM
NUM 50
NUM ,1
NUM 50,1

The **NUM** command enables BASIC XE's automatic line numbering ability. This facility can increase your program entry speed because it puts in the program line numbers for you. If no **start** or **inc** is given (first example), **NUM** will start numbering from the last line number currently in the program in increments of 10. If there is no current program, **NUM** will start with line number 10. If the starting line number alone is given (second example), **NUM** will start numbering from that line number in increments of 10. If the increment alone is given (third example), **NUM** will start numbering from the last line currently in the program, in increments of **inc**. If both the starting line number and the increment are given (last example), **NUM** will start numbering from the given line number in increments of **inc**. **Note:** neither **start** nor **inc** may be 0.

Four things cause the automatic line numbering to stop:

1) If you press <RETURN> immediately following the line number.
2) If BASIC XE encounters a syntax error on a program line you type in.
3) If the line number the automatic numberer would use already exists.
4) If the automatic numberer would generate a number larger than 32767.

Note: using **NUM** in Deferred Mode always returns you to Direct Mode.

LIST (L.)

Format: LIST [lineno1][,[lineno2]]
Examples: LIST
LIST 10
LIST 10,100
LIST 10,

Note: this section covers only the editing uses of **LIST**. For its program saving uses, see the **Storing and Retrieving Your Program** chapter.

LIST causes the program currently in memory to be displayed so that you can edit or study it. If **LIST** is used alone (without lineno1 or 2), the entire program is displayed (first example). If you follow it with a single line number, only that line will be displayed (second example). If you specify two line numbers (separated by commas), lines **lineno1** through **lineno2** will be **LIST**ed (third example). If you give the starting line number, a comma, and no ending line number, the ending line number is assumed to be the last line in the program (last example).

Note: You can control the automatic indention of structured statements (**FOR**, **WHILE**, etc.) when they are **LIST**ed using **SET 12,aexp**. You can also control the casification using **SET 5,aexp**. See **SET** for more info.

DEL

Format: DEL lineno1[,lineno2]
Examples: DEL 100
DEL 1000,1999

DEL deletes program lines currently in memory. If a single line number is given, only that line will be deleted (first example). If two line numbers are given, lines **lineno1** through **lineno2** (inclusive) will be deleted (second example).

RENUM

Format: RENUM [start][,inc]
Examples: RENUM
RENUM 100
RENUM ,20
RENUM 1000,5

RENUM renumbers the program in memory, using **start** as the starting line number, and **inc** as the increment between line numbers. If **start** is not specified, 10 is used. If **inc** is not specified, an increment of 10 is assumed. **Note:** neither **start** nor **inc** may be 0.

All line number references (e.g., in GOTOs, GOSUBs, etc.) are also renumbered if they are numeric constants. Line number expressions (e.g., GOTO 10*A) will not be renumbered.

Caution: if you are RUNning a program in **FAST** mode, a **RENUM** in that program will do nothing.

Caution: If you use **LIST** in Deferred Mode (i.e., in a program) the line number values you want to list will not be renumbered by **RENUM**.

Caution: **RENUM** will not renumber an absolute line number after a line number expressed as an expression. If you RENUM the statement

```
10 On X Gosub 100,3*Y,200
```

the 100 will be renumbered, but the 200 will not since it follows a line number expression (3*Y). This situation is possible only in the **ON** statement.

Warning: If you have a reference to a line number that does not yet exist (e.g. a **GOTO 50** when line 50 doesn't exist), **RENUM** will not renumber that reference. After the **RENUM**bering, however, the non-existent line number might exist, thus making the reference valid, but it will most likely not refer to the program line you want it to.

REM (R.)

Format: REM text
Examples: REM This is a remark
10 REM Routine to calculate X
20 GOSUB 300 : REM Find Totals

REM stands for "remark" and is used to put comments into a program. This command and the text following it on the same line are ignored by the computer. However, it is included in a **LIST** along with the other numbered lines. Since all characters following a **REM** are treated as part of the **REM**ark, no statements following it (on the same program line) will be executed.

Space For Your Notes

Storing and Retrieving Your Program

BASIC XE allows you to store your programs in either of two formats - as ATASCII text, or as the tokenized gibberish internal to BASIC XE. **LIST** and **ENTER** perform program I/O using the first format, while **SAVE** and **LOAD**, and **CSAVE** and **CLOAD** use the second. The reason the tokenized format is offered is that it is generally more compact than the ATASCII format and always cuts down on disk/cassette use and I/O time.

LIST (L.)

```
Format:    LIST "filespec"[,lineno1][,[lineno2]]
Examples:  LIST "C:"
           LIST "D:DEMO.LIS"
           LIST "P:",20,100
```

LIST allows you to write out the ATASCII text version of the program in memory. As evident from the examples, **filespec** may refer to any device. You may add any of the line number specifications (described in the previous chapter's discussion of **LIST**) to **LIST** only a portion of your program to **filespec**.

Note: the quotes around **filespec** are required by **LIST**, unless of course a string variable is used.

ENTER (E.)

```
Format:    ENTER "filespec"
Examples:  ENTER "C:"
           ENTER "D2:DEMO.LIS"
```

The **ENTER** command allows you to read in a program you have saved using the **LIST** command, and will not work with programs which have been **SAVE**d or **CSAVE**d. To use this command, you simply need to give the filespec of the program. **Note:** whereas both **LOAD** and **CLOAD** clear the program memory space before reading in the new program, **ENTER** does not, and so is useful when trying to merge programs together.

Bonus: You can modify what BASIC XE does after completing an **ENTER** using the **SET 9,aexp** command (see **SET** for more info).

SAVE (S.)

Format: SAVE "filespec"
Examples: SAVE "D:TEST.BXE"
SAVE "C:"

SAVE allows you to save the tokenized form of a BASIC XE program to any device. A file saved using this command may then be read back into program memory using LOAD or loaded and immediately executed using the RUN command.

LOAD (LO.)

Format: LOAD "filespec"
Examples: LOAD "D1:GAME1.BXE"
100 LOAD "C:"

LOAD allows you to load the SAVEd version of a program into memory from any device. It will not work with programs saved using LIST or CSAVE.

CSAVE (CS.)

Format: CSAVE
Examples: CSAVE
100 CSAVE
100 CS.

CSAVE is used to save the tokenized version of a program. The difference between CSAVE and SAVE "C:" is that CSAVE leaves shorter inter-record gaps and so makes cassette I/O faster. On entering CSAVE two bells sound to indicate that the PLAY and RECORD buttons must be pressed, followed by <RETURN>. Do not, however, press these buttons until the tape has been positioned. Note: tapes saved using the two commands SAVE and CSAVE are not compatible. Note: due to a flaw in the Atari OS ROMs (not BASIC XE), it may be necessary on some machines to enter an LPRINT before using CSAVE, otherwise it may not work properly. For specific instructions on how to connect and operate the hardware, cue the tape, etc., see the Atari 410 or 1010 Program Recorder Manual.

CLOAD

Format: CLOAD
Examples: CLOAD
100 CLOAD

This command can be used in either Direct or Deferred Mode to load a program from cassette tape, and may be used only with programs which have been CSAVEd. On entering CLOAD, one bell sounds to indicate that the PLAY button needs to be pressed, followed by <RETURN>. However, do not press PLAY until the tape has been positioned. Specific instructions for CLOADing a program are contained in the Atari 410 or 1010 Program Recorder Manual.

Making Your Program Stop and Go

The statements discussed in this chapter enable and control the execution of your BASIC XE program. They are **RUN, END, FAST, STOP, CONT, TRACE,** and **TRACEOFF.**

RUN

Format: RUN ["filespec"]
Examples: RUN
100 RUN "D:MENU"

This command causes BASIC XE to begin executing a program. If filespec is not specified, the current RAM-resident program is executed; otherwise BASIC XE retrieves the tokenized program form the specified file and then executes it. Before execution begins, **RUN** sets all avars to zero, unDIMensions all mvars, svars, and savars, **CLOSE**s all open files (channels), and turns off all **SOUND**s. If an error occurs while your program is **RUN**ning, execution will halt and an error message will be displayed (unless the error has been **TRAP**ped).

Although **RUN** without a filespec is most frequently used in Direct Mode, it can also be used in Deferred mode. For example, **RUN** the following program (press <BREAK> to exit):

```
10 Print "Continuous RUNning"
20 Run
```

Note: **RUN** must be the last (or only) command on a program line when used in Deferred Mode.

If you want to begin program execution somewhere other than at the first program line, use **GOTO** in Direct Mode. **Caution:** variables are neither cleared nor initialized by **GOTO.**

END

Format: END
Examples: END
4000 END

END is used to terminate the execution of a program. In addition to this, it also closes all files (channels), silences any sounds, and turns off P/M's (if they were turned on via **PMG.**). It does not change the graphics mode, however. **END** is not required in most programs because BASIC XE automatically closes all files and silences any sounds after the last program line has executed.

Note: if you have any subroutines following the main program you should put an **END** at the end of the main program, or the subroutines may be executed as part of the main program.

END may also be used in Direct mode to close files, silence sounds, and turn off P/M's.

FAST

Format: FAST
Examples: FAST
100 FAST

During normal program execution BASIC XE must search from the beginning of your program for a specified line number whenever it encounters a **GOTO, GOSUB, FOR,** or **WHILE** (this is how most other BASICs do it too). However, you can change this by using the **FAST** command. When BASIC XE sees **FAST**, it does a precompile of the program currently in memory. During the precompile BASIC XE changes every line number to the address of that line in memory. Then, whenever a **GOTO, GOSUB, FOR,** or **WHILE** is executed, no line number search is needed, since BASIC XE can simply jump directly to the specified line's address.

Note: if the lineno used in the **GOTO** or **GOSUB** is not a constant (i.e., is a variable or an expression), that lineno will not be affected by **FAST**, and so will execute at normal speed.

Note: the following statements and situations will terminate **FAST** mode execution:

DEL
ENTER
EXTEND
LIST
LOAD
LVAR
RUN "filespec"
SAVE
returning to Direct Mode.

Caution: when you use **FAST** in Deferred Mode, it must precede your first **GOSUB, FOR, CALL, WHILE,** and/or **LOCAL**. We recommend that you use it as the first statement in your program.

Caution: if you are using **ENTER** to create program overlays, you will notice that the notes and caution above seemingly combine to preclude the possibility of **ENTER**ed overlays executing in **FAST** mode. There is only one way to get around this: the main program (the part that calls the overlays) cannot be in a loop, subroutine, or local region when it **ENTER**s the overlay. If you insure this, you may then make **FAST** the first statement in your overlay without creating problems.

STOP

Format: STOP
Examples: 100 STOP

When you use the **STOP** command in Deferred Mode in a program, BASIC XE displays the message "Stopped at line lineno", terminates program execution, and returns to Direct Mode. **STOP** does not close files or turn off sounds (as does **END**), so the program can be resumed by typing **CONT**. This can be very useful in error handling. For more information on this, see the **Handling Errors** chapter. When used in Direct Mode, **STOP** simply displays "Stopped", and returns to Direct Mode.

CONT

Format: CONT
Examples: CONT
100 CONT

In Direct Mode, **CONT** resumes program execution which has been interrupted by a **STOP** statement, a <BREAK> key abort, or an error. **Caution:** execution resumes on the line following the halt, so any statements following the halt, but on the same program line, will not be executed.

In Deferred Mode, **CONT** may be used for error handling. For these uses, see the **Handling Errors** chapter.

TRACE / TRACEOFF

Formats: TRACE
TRACEOFF
Examples: 100 TRACE
TRACEOFF

These statements are used to enable or disable the line number trace facility of BASIC XE. When in **TRACE** mode, the line number of a line about to be executed is displayed on the screen, surrounded by brackets ([]).

Exceptions: The first line of a program cannot be **TRACE**d, nor can the target line of a **GOTO**, **GOSUB**, or **CALL**, or the looping line of a **FOR** or **WHILE**.

Note: a statement issued in Direct Mode is **TRACE**d as having line number 32768.

TRACEOFF is used to turn **TRACE**ing off once it has been enabled.

Space For Your Notes

Configuring the BASIC XE System

The statements and functions in this chapter allow you to change how BASIC XE will function, as well as find out the current configuration. The statements discussed are **SET**, **LOMEM**, **CLR**, **LVAR** and **EXTEND**, and the functions are **SYS** and **FRE**.

SET

Format: SET aexp1,aexp2

The **SET** statement allows you to change a variety of BASIC XE system-level functions. **aexp1** is the function you wish to change, and **aexp2** is the value to alter the function. The table following summarizes these **SET** parameters (default values are given in parentheses):

aexp1		aexp2	Meaning
0	(0)	0	<BREAK> key functions normally. Note: Returning to Direct Mode does a **SET 0,0**.
		1	<BREAK> causes a **TRAP**able error (#1) to occur.
		128	<BREAK>s are ignored by BASIC XE. Other subsystems (F: for example), however, will still recognize <BREAK>s.
1	(10)	1...128	Tab stop setting for the comma in PRINT statements.
2	(63)	0...255	Prompt character for INPUT (default is "?").
3	(0)	0	**FOR** loops execute at least once (ala Atari BASIC).
		1	**FOR** loops may execute zero times (ANSI standard).
4	(1)	0	Instead of reprompting, a **TRAP**able error (#8) occurs.
		1	On a multiple variable **INPUT**, if the user enters too few items, he is reprompted (e.g., with "??")
5	(1)	0	BASIC XE acts like Atari BASIC in that it is sensitive to character case on program entry (either type-in or **ENTER**). Lowercase and/or inverse characters cause syntax errors, except when used in **REM** , **DATA**, or string constants.
		1	BASIC XE converts text to a nice, readable format upon entry. Keywords and variable names are capitalized, while **REM** text, **DATA** items, and string constants remain unchanged.
6	(0)	0	Print error messages along with error numbers.
		1	Print only error numbers (ala Atari BASIC).
7	(0)	0	P/M's that move vertically to the edge of the screen roll off the edge and are lost.
		1	P/M's wrap around from top to bottom and visa versa.

aexp1		aexp2	Meaning
8	(1)	0	Don't push (PHA) the number of parameters to a USR call on the stack (advantage: some assembly language subroutines not expecting parameters may be called by a simple USR).
		1	Do push the count of parameters, ala Atari BASIC.
9	(0)	0	ENTER returns to Direct Mode on completion.
		1	End-Of-ENTER creates a TRAPable error (#32).
10	(0)	0	The four missiles act independently.
		1	The four missiles are grouped together for movement purposes. However, their widths and colors remain independent.
11	(40)	1...255	BASIC XE will automatically DIM a string to this size if you do not DIMension it yourself.
		0	BASIC XE works like Atari BASIC.
12	(1)	0	The LIST program formatter does not indent when you use structured statements (FOR, WHILE, etc.).
		1	LIST indents when you use structured statements.
13	(1)	0	VAL produces an error (#18) if you use a hex digit string.
		1	VAL will turn hex digit strings into numbers, provided that the string begins with a "$".
14	(0)	0	PRINT USING truncates numbers when they contain more digits than specified in the format.
		1	This situation produces a TRAPable error (#23).
15	(0)	0	In EXTENDed Mode only, ADR("string") will produce an error 3.
		1	ADR("string") will always return the address of string.

f SYS

Format: SYS(aexp)
Example: 100 IF SYS(0)=0 THEN SET 0,128

The SYS function is used to find out the status of a BASIC XE system function alterable using SET. aexp is the number of the system function as defined in the previous section.

LOMEM

Format: LOMEM addr
Example: LOMEM DPEEK(128)+1024

LOMEM is used to reserve space below the normal program space. You could then use this space for screen display information or assembly language routines. The usefulness of this may be limited, though, since there are other more usable reserved areas available. Caution: **LOMEM** wipes out any user program currently in memory.

CLR

Format: CLR
Example: 200 CLR

The **CLR** statement clears the values in the Variable Value Table and unDIMensions all svars, savars, and mvars. It does not clear the Variable Name Table (only **NEW** does), so all the names remain. If you wish to use an svar, savar, or mvar after using **CLR**, you must reDIMension it first.

f FRE

Format: FRE(aexp)
Examples: PRINT FRE(0)
100 IF FRE(0)<1000 THEN PRINT "Memory Critical"

The **FRE** function returns the number of of RAM bytes left for your use. Normally **FRE(0)** returns the total amount of memory left, but if you have used the **EXTEND** statement, **FRE(0)** returns the amount of data space left, and **FRE(1)** returns the amount of program space left in the extended memory area.

LVAR (LV.)

Format: LVAR ["filespec"]
Example: LVAR "P:"

LVAR will list all variables currently in use to filespec. Each variable is followed by a list of the lines on which that variable is used. The example above will list the variables to the printer. If filespec is not specified, **LVAR** lists to the screen.

Note: svars and savars are denoted by a trailing "$", and mvars by a trailing "(".

Warning: **LVAR** must be the last (or only) statement on a program line.

EXTEND

Format: EXTEND

Until you use the **EXTEND** command with a 130XE, BASIC XE operates very much like Atari BASIC. From the viewpoint of most programs, BASIC XE in 'normal' mode is Atari BASIC. Faster, and with many additional capabilities, but very memory compatible.

EXTEND tells BASIC XE to switch from Atari BASIC 'normal' mode to 'extended' mode. In extended mode, BASIC XE programs reside in the 'extra' 64K bytes of a 130 XE, labeled 'extended memory' in the second diagram of Appendix B. Programs can use up all 64K bytes of the extended memory without intruding upon the data space (for strings, arrays, etc.) in main memory (again, see Appendix B).

You may use the **EXTEND** command in Direct Mode at any time--either when you have no program in memory or after a program is in place. **EXTEND** will transfer any program in main memory to the extended memory. Once in extended mode, the only ways to return to 'normal' mode are to use the **NEW** command or to **LOAD** a program which was **SAVE**d in normal mode.

On the other hand, you will automatically enter extended mode if you **LOAD** a program that was **SAVE**d from extended mode. Once you have **EXTEND**ed a program, you can restore it to normal mode only by **LIST**ing and re-**ENTER**ing it.

Note: **EXTEND** can only be used in Direct Mode, never in a program.

Note: You must be using an Atari 130XE computer (or equivalent) for this command to work. If BASIC XE cannot find the extended memory banks, you will see an Error 60, "Extended Memory Not Available".

Note: BASIC XE follows recently established Atari Corporation guidelines when it uses the extended memory. In particular, if the extended memory is already in use (e.g., by Atari DOS 2.5's RamDisk), BASIC XE will not let you **EXTEND** your program and will give you an Error 60, as above. Early versions of DOS 2.5, as well as other programs, may not yet follow these new guidelines, so be sure the extended memory is available before using the **EXTEND** command.

Technical Note: BASIC XE fills the extended memory with your program from the 'bottom' up. Referring to the second diagram in Appendix B, this means that approximately the first 16K bytes of your program will go in Bank 0. The next 16K bytes go in Bank 1, etc. These numbers are not exact, because (1) BASIC XE always maintains a minimum of $100 bytes of free space in each bank, and (2) BASIC XE never breaks program lines between banks.

Still, if you subtract about $400 from the value returned by **FRE(1)**, you will have a lower bound on the amount of space left in extended memory. Then you could, for example, use bank 3 to store miscellaneous data, provided that **FRE(1)-$400** shows at least 16K bytes left. See appendix D for details, or see your Atari 130XE owner's manual for information on how the hardware side of the bank selection works.

Exiting BASIC XE

The following two commands, **DOS** and **BYE**, are used to leave BASIC XE to use some other utility.

DOS (CP)

Format: DOS

DOS is used to go from BASIC XE to the Disk Operating System (DOS). If you have not booted a DOS into memory, the computer will go into Self-Test Mode and you must press <SYSTEM RESET> to return to BASIC XE. If you have botted with a DOS, control passes to DOS. To return to BASIC XE, type "CAR" if you are using DOS XL, or press "B" if you're using Atari DOS.

DOS is usually used in Direct Mode, but it may be used in a program as well. For more details on this, see your DOS manual.

Note: CP (command processor) is exactly equivalent to **DOS**.

BYE (B.)

Format: BYE

The function of **BYE** is to exit BASIC XE and go directly into your computer's Self-Test Mode. To return to BASIC XE, press <SYSTEM RESET>.

Space For Your Notes

Introducing Atari I/O

The Atari Personal Computers consider everything except the guts of the computer (i.e. the RAM, ROM, and processing chips) to be external devices - for example, the Keyboard and Screen Editor. Some of the other devices are Disk Drive, Program Recorder (cassette), and Printer. The following is a list of the devices, ordered according to the device specifier. For some devices the specifier alone is needed as "filespec", while others require both the specifier and a file name:

C: The Program Recorder - handles both Input and Output. You can use the recorder as either an input or output device, but never as both simultaneously.

D1: - D8: Disk Drive(s) - handle both input and Output. Unlike C:, disk drives can be used for input and output simultaneously. Floppy disks are organized into a group of files, so you are required to give a file name along with the device specifier (see your DOS manual for more information). Note: if you use D: without a drive number, D1: is assumed.

E: Screen Editor - handles both Input and Output. The screen editor simulates a text editor/word processor using the keyboard as input and the display (TV or Monitor) as output. This is the editor you use when typing in a BASIC XE program. When you specify no channel while doing I/O, E: is used because the I/O channel number defaults to 0, which is the channel BASIC XE opens for E:.

K: Keyboard - handles Input only. This allows you access to the keyboard without using E:.

P: Parallel Port on the 850 Module - handles Output only, Usually P: is used for a parallel printer, so it has come to mean "Printer" as well as "Parallel Port".

R1: - R4: The RS-232 Serial Ports on the 850 Module - handle both Input and Output. These devices enable the Atari system to interface to RS-232 compatible serial devices like terminals, plotters, and modems. Note: if you use R: without a device number, R1: is assumed.

S: The Screen Display (TV or Monitor) - handles both Input and Output. This device allows you to do either character or graphics I/O on the screen display. The cursor is used to address a screen position.

Each of these devices is used for I/O of some type, although only a few of them can do both input and output (you wouldn't want to input data from a Printer). Because they work differently, each device has to tell the computer how it operates. This done through the use of a device handler. A device handler for a given device gives information on how the computer should input and output data for that device.

One of the sub-systems in the computer is the Central Input/Output (CIO) processor. It is CIO's job to find out if the device you specify exists, and then look up I/O information in that device's handler. This makes it easy for you, since you don't need to know anything about given handler. To let CIO know that a device exists (i.e., is available for I/O) you need to OPEN the device on one of the CIO's

eight channels (numbered 0-7). When you want to do I/O involving the OPENed device, you must then use the channel number instead of the device name.

When you see "filespec" in the following sections, it refers simply to the device (and file name in the case of D:) in a character string. The string may be either a string constant, an svar, or an savar element.

If you use channel #7, it will prevent LPRINT or some of the other BASIC XE I/O statements from being performed.

OPEN

Format: OPEN #chan, aexp1, aexp2, "filespec"
Examples: 100 OPEN #2,8,0,A$
OPEN #4,4,0,"D:INPUT.TXT"

As mentioned above, a device must be OPENed on a specific channel before it can be accessed. This "opening" process links a specific channel to the appropriate device handler, initializes any CIO-related control variables, and passes any device-specific options to the device handler. The parameters for the OPEN command are defined as follows:

chan This is the number of the channel which you want to associate with the device filespec. Also, this is the number you use when you later want to do I/O involving the specified device (using INPUT, PRINT, etc.).

aexp1 This is the I/O mode you want to associate with the above channel. The numeric codes are described in the following table:

aexp1	Meaning
4	Input Only
6	Read Disk Directory Only
8	Output Only
9	Output Append
12	Input and Output

Note: other modes may exist for special devices or extensions to a device.

aexp2 Device-dependent auxiliary code. See your device manual to see if it uses this number. If not, use a zero.

filespec The device (and file name, if required) you want to be associated with the specified channel.

CLOSE (CL.)

Format: CLOSE #chan
Examples: CLOSE #4
100 CLOSE #1

CLOSE is used to close a CIO channel which has been previously OPENed to allow I/O on some device. After you CLOSE a channel, you can then reOPEN it to some other device, and thus associate that channel number with a different device.

Note: you should CLOSE all channels you have OPENed when you are finished using them.

PRINT (PR. or ?)

Format: PRINT [#chan] [| ; | exp...][| , |]
| , | | ; |

Examples: PRINT
PRINT X,Y,Z;A$
100 PRINT "The value of X is ";X
100 PRINT "Commas","cause","tabs"
100 PRINT #3,A$
100 PRINT #4;"$";HEX$(X);" is ";X

PRINT is used in either Direct or Deferred Mode to output data. In Direct Mode, it prints whatever exp information is given. In the second example, the screen will display the current values of X,Y,Z, and A$. In the fifth example, A$ is PRINTed out to the device associated with channel 3.

The comma option causes tabbing to the next tab location. Several commas in a row cause several tab jumps. To set the tab spacing caused by the use of a comma, use SET 1,aexp (see SET for more info).

A semicolon causes the next exp to be output immediately after the preceding exp without spacing or tabbing. Therefore, in the sixth example spaces surround the 'is' so that it and the values of X will not butt up against each other.

If no comma or semicolon is used at the end of a PRINT statement, then a <RETURN> is output and the next PRINT will start on the following line.

Note: numbers smaller than 0.01 or with more than 10 significant digits will be PRINTed in scientific notation.

INPUT (I.)

```
Format:    INPUT | [#chan,] | var1 [,var2...]
                 |["string"]|
Examples:  INPUT X
           100 INPUT SA$(4;)
           100 INPUT X,Y,Z(4),B$
           100 INPUT #4,A$(5,9)
           100 INPUT "SS#,Name>> ",Ssnum(X),Names$(X;)
```

INPUT is used to input various data and store it directly into variables. The first data element **INPUT**ted will be stored in **var1**, the second in **var2**, and so on. If you are **INPUT**ting more than one arithmetic variable, the numeric data elements may be entered on a single line if they are separated by commas, or on separate lines, each followed by a <RETURN>. In the latter case, BASIC XE will prompt with a double question mark to indicate that more input is needed. When **INPUT**ting a group of strings, each must be typed on a line by itself, or as the last item on the line when combined with numeric input.

Note: you can make BASIC XE produce a **TRAP**able error instead of the double prompt by using **SET 4,**aexp. Also, you can change the default question mark (?) prompt to any character using **SET 2,**aexp (see **SET** for more info).

The fifth example above shows off one of the most powerful additions to **INPUT**. If a literal string immediately follows the **INPUT**, that string will be used as the prompt, thus allowing you to create prompts that are more explanatory than the standard "?".

We strongly recommend that:

1) no more than one variable be used on each **INPUT** line.
2) **INPUT** and **PRINT** should not be used for disk data file access (**RGET** and **RPUT** are suggested instead).

Bonus: as you can see from the third and fourth examples above, you can **INPUT** directly in mvar elements and/or substrings. This addition (not in Atari BASIC) can be extremely useful and make your programs very efficient.

PUT (PU.)

```
Format:     PUT #chan,aexp
Examples:   PUT #6,ASC("A")
            100 PUT #0,4*13
```

PUT is used to output a single byte of data to an open channel. The data output is **aexp**, and it is output to channel **chan**.

GET

```
Format:   GET #channel, avar
Example:  100 GET #0, X
```

GET is used to input one byte of data from an open channel. This byte of information is stored in **avar**.

LPRINT (LP.)

```
Format:   LPRINT [exp][|;|exp...][|;|]
                     |,|         |,|
Example:  LPRINT "Calculation of X squared:"
```

LPRINT causes BASIC XE to output data on the printer rather than on the screen. It can be used in either Direct or Deferred Mode, and requires neither device specifier nor **OPEN** or **CLOSE** statement.

Caution: LPRINT cannot be used successfully with most printers when a trailing comma or semicolon is used. If advanced printing capabilities are required, we recommend using **PRINT #** on a channel previously **OPEN**ed to the printer (P:).

Note: the semicolon and comma options are discussed in the **PRINT** section of this chapter.

Note: although **LPRINT** may be used with **USING** just like **PRINT**, we recommend using **PRINT #x; USING** instead.

TAB

Format: TAB [#chan,] aexp
Examples: TAB #2,20
100 TAB 12

TAB outputs spaces to the device specified by chan (or the screen if chan is not specified) up to column aexp. The first column is numbered 0.

Note: the column count is kept for each device and is reset to zero each time a carriage return is output to that device. The count is kept in Aux6 of the IOCB (See OS documentation).

Note: if aexp is less than the current column count, a <RETURN> is output and then spaces are put out up to column aexp.

f TAB

Format: TAB(aexp)
Example: PRINT #3;"columns:"TAB(20);20;TAB(30);30

The TAB function's effect is identical to that of the TAB statement (see above). The difference is that imbedding a TAB function in a PRINT USING or PRINT can simplify your programming task greatly. The TAB function will output sufficient spaces so that the next item will print in the column specified (only if the TAB(aexp) is followed by a semicolon, though).

Note: if aexp is less than the current column count, a carriage return is output and then spaces are output up to column aexp.

Caution: the TAB function will output spaces on some device whenever it is used; therefore, it should be used only in PRINT or PRINT USING statements.

Advanced Data Input/Output

The statements in this chapter deal with special applications or advanced concepts of data I/O. Unless you are already familiar with these or similar statements (i.e. if you've used BASIC XL), we suggest that you play with them a little just to get a feel for what they can and can't do.

PRINT USING

Format: PRINT [#chan|;|,] USING sexp, exp1 [,exp2...]

PRINT USING allows you to specify a format for the data you wish to output. **sexp** is the string which defines the format you wish to use, and is made up of one or more format fields. Each format field tells how one of the exps which follow **sexp** is to be printed. The first field specifies the first exp's format, the second field specifies the second exp's, and so on. The valid format field characters are **# & * + $, . % !** and **/** (each will be explained separately in just a moment). Non-format characters terminate a format field and are printed as they appear.

Note: the comma (,) and semicolon (;) spacing options of **PRINT** are overridden in the expression list of **PRINT USING**, but apply after chan if it is used (i.e. ',' produces a tab, and ';' produces no spacing).

Warning: **sexp** must contain at least one valid format field, otherwise BASIC XE will print **sexp** repeatedly as it searches for a format field.

Numeric Formats: the characters for formatting numbers are:

# Blank Fill	, Insert a Comma
& Zero Fill	+ Sign (+/-) pre/postfix
* Asterisk Fill	- Sign (- only) pre/postfix
. Decimal Point	$ Dollar Sign prefix

 # & and *: if there are fewer digits in the output number than specified in the format, then the digits are right justified in the field and prefixed with the proper fill character. If there are more digits in the output number than specified in the format, then the rightmost digit(s) of the number which fit in the field format are displayed (see last example). The following table illustrates these capabilities and limits (bars have been placed around the output so that you may visualize the field boundaries):

Value	Format	Output
123	####	| 123|
123	&&&&	|0123|
123	****	|*123|
1234	####	|1234|
12345	####	|2345|

Note: if you don't want numbers truncated, you can use **SET 14,1**. BASIC XE will then force a **TRAP**able error (#23) rather than truncate the number.

. (period): a period in the format field indicates that a decimal point is to be printed at that location in the number. All digit positions in the format that follow the decimal point are filled with digits. If the output number contains fewer fractional digits than specified in the format, then zeroes are printed in the extra positions. If the output number contains more fractional digits than indicated in the format, then the output number is rounded so that there are the specified number of fractional digits. Note: a second decimal point within a single format is treated as a non-format character, and so terminates the format field. Here are some examples:

Value	Format	Output
12.488	###.##	12.49
123.4	###.##	123.40
2.35	**.**.	*2.35.

, (comma): a comma in the format field indicates that a comma is to be printed at that location in the output number. If the format specifies that a comma should be printed at a position that is preceded only by fill characters (#,&,*), then the appropriate fill character will be printed instead of the comma. Note: the comma is a valid format character only to the left of the decimal point (if a decimal point is used); when a comma appears to the right of a decimal point, it becomes a non-format character and terminates the format field. Here are some examples:

Value	Format	Output
5216	##,###	5,216
3	*,****	*****3
4175	#,###.	4,175.

+ and -: a plus sign in a format field indicates that the sign of the output number is to be printed (+ if positive, - if negative). A minus sign indicates that a minus sign (-) is to be printed if the output number is negative and a blank if the output number is positive.

The signs may be fixed or floating prefixes, or fixed postfixes. When used as fixed prefixes, the sign format character be the first character in a format field:

Value	Format	Output
43.7	+###.##	+ 43.70
-43.7	+###.##	- 43.70
23.58	-&&&.&&	023.58
-23.58	-&&&.&&	-023.58

Floating signs must start in the first format position and occupy all positions up to the decimal point. This causes the sign to be printed immediately before the first digit rather than in a fixed location. Each sign after the first also represents a blank-fill digit position:

Value	Format	Output
3.75	+++.##	+3.75
3.75	---.##	3.75
-3.75	---.##	-3.75

A trailing sign may appear only after a decimal point and as the last character in the format field. It terminates the format and prints the appropriate sign (or blank):

Value	Format	Output
43.17	***.**+	*43.17+
43.17	&&&.&&-	043.17
-43.17	###.##+	43.17-

$ (dollar sign): a dollar sign in a format field indicates that a $ is to be used as a fixed or floating prefix to the output number. A fixed dollar sign must be either the first or second character in the format field (second only if the first is a + or - used as a fixed sign prefix):

Value	Format	Output
34.2	$##.##	$34.20
34.2	+$##.##	+$34.20
34.2	-$##.##	$34.20
-34.2	+$###.##	-$ 34.20

Floating dollar signs must start as either the first or second (second for reasons outlined above) character in the format field and continue to the decimal point. Each dollar sign after the first also represents a blank-fill digit position:

Value	Format	Output
34.2	$$$$$.##	$34.20
34.2	+$$$$$.##	+ $34.20
-72692.41	$$$,$$$.##+	$72,692.41-

Note: There may be only one floating character per format field.

Warning: using +, - or $ in other than proper positions will give strange results.

String Formats: the format characters for strings are as follows:

```
% indicates the string is to be right justified.
! indicates the string is to be left justified.
```

If there are more characters in the string than in the format field, then the string is truncated. Following are examples of string formatting:

String	Format	Output
"BASIC XE"	%%%%%%%%%	BASIC XE
"BASIC XE"	!!!!!!!!!	BASIC XE
"BASIC XE"	%%%%%	BASIC
"BASIC XE"	!!!!!	BASIC

Embedding Characters: the slash character (/) does not terminate the format field but will cause the next character to be printed as is, thus allowing you to insert non-format characters in the middle of a format field, as in the following examples:

```
   Value          Format          Output
4084463099   (###/)###/-####   |(408)446-3099|
"OSS"        %/.%/.%/.         |O.S.S.|
```

Bonus: if there are more expressions in the list than there are format fields, the format fields will be reused. For example,

```
PRINT USING "####",25,19,7
```

will output

```
|  25  19   7|
```

NORMAL / INVERSE

Format: NORMAL
INVERSE

Examples: NORMAL
100 NORMAL
150 INVERSE

NORMAL and **INVERSE** allow you to change the video presentation of all **PRINT**s, **LPRINT**s, and **PRINT USING**s. Anything you display after a **NORMAL** will be output just as it appears in your program, while anything you display after using **INVERSE** will be converted to inverse video. In this case, characters that were previously in inverse video will appear in normal video.

Note: BASIC XE returns to **NORMAL** display whenever you return to Direct Mode or re**RUN** a program from within itself.

BPUT

Format: BPUT #chan, aexp1, aexp2 [,bank]

BPUT outputs a block of data to the device OPENed on channel chan. The block of data starts at address aexp1, and is aexp2 bytes long. You may also select an optional bank number if you're in EXTENDed mode (see EXTEND for more info).

Note: aexp1 the address may be a memory address, or the address of a string (found using ADR).

The following example writes out an entire mode 8 graphics screen directly from screen memory:

```
100 Graphics 8:Addr=Dpeek($58)
110 Print "Filling Screen..."
120 For Sbyte=0 To (40*160)-1:Rem "fill screen"
130   Poke Addr+Sbyte,Random(256)
140 Next Sbyte
150 Print "Done Filling. Now BPUTting..."
160 Close #1:Open #1,8,0,"D:MODE8.SCR":Rem "ready to BPUT"
170 Bput #1,Addr,40*160
180 Close #1
190 Print "Finished BPUTting"
200 End
```

Note: nothing is written to the file which indicates the length of the data written. We suggest that you write fixed-length data to make the rereading process simpler.

BGET

Format: BGET #chan, aexp1, aexp2 [,bank]

BGET gets aexp2 bytes from the device OPENed on channel chan, and stores them starting at address aexp1. As with BPUT, aexp1 may be the address of a string; in this case BGET does not change the length of the string - this is your responsibility. You may also select an optional bank number if you're in EXTENDed mode (see EXTEND for more info).

The following example will read in an entire mode 8 graphics screen directly into screen memory:

```
100 Graphics 8:Addr=Dpeek($58)
110 Close #1:Open #1,4,0,"D:MODE8.SCR":Rem "ready to BGET"
120 Print "Now BGETting..."
130 Bget #1,Addr,40*160
140 Close #1
150 Print "Finished BGETting"
160 End
```

Note: no error checking is done on the address or length so care must be taken when using this statement, lest you wipe out part of DOS or your BASIC XE program.

RPUT

Format: RPUT #chan, exp [,exp...]

RPUT allows you to output fixed-length records to the device OPENed on channel chan. Each exp constitutes one field element in the record. An arithmetic field consists of one byte which indicates an arithmetic data type, and 6 BCD floating point bytes of data. A string field consists of one byte which indicates a string data type, 2 bytes of LEN length, 2 bytes of DIM length, and then DIM length bytes of data. All this really means is that you can't INPUT data which has been RPUTted, since more than just the data is RPUT.

The following example RPUTs 20 records of the form "Name", "Address", "City", "State", Zip, Phone:

```
100 Dim Names$(20,30),Addrs$(20,30),Cities$(20,20),States$(20,2)
110 Dim Zips(20),Phones(20)
120 Close #1:Open #1,8,0,"D:FRIENDS.DAT"
130 For Recnum=1 To 20
140   Input "Name>>     ",Names$(Recnum;)
150   Input "Address>> ",Addrs$(Recnum;)
160   Input "City>>     ",Cities$(Recnum;)
170   Input "State>>    ",States$(Recnum;)
180   Input "Zip>>      ",Zips(Recnum)
190   Input "Phone>>    ",Phones(Recnum)
200   Print :Print "I've got:"
210   Print Names$(Recnum;):Print Addrs$(Recnum;)
220   Print Cities$(Recnum;);", ";States$(Recnum;);" ";Zips(Recnum)
230   Print  Using "(###)###/-####",Phones(Recnum)
240   Print :Input "Ok to save (Y/N)? ",Ans$
250   If (Ans$="Y") Or (Ans$="y"):Rem "do RPUT"
260     Rput #1,Names$(Recnum;),Addrs$(Recnum;),Cities$(Recnum;)
270     Rput #1,States$(Recnum;),Zips(Recnum),Phones(Recnum)
280   Else :Print "Re-enter record":Goto 140
290   Endif
300 Next Recnum
310 Close #1:Print :Print "All Done"
320 End
```

RGET

Format: RGET #chan, var [,var...]

RGET allows you to retrieve fixed-length records from the device OPENed on channel chan, and assign the values to string or arithmetic variables. Note: the input data and the variable into which the data is stored must be of the same type (i.e. they must both be string or both be arithmetic).

Note: when the data type is string, then the DIMensioned length of the data string must be equal to the DIMensioned length of the svar. Once the data string has been assigned to the svar, RGET sets the LEN length of the svar to the actual length of the inputted data string (not the DIM length of the data string).

Warning: you may not RGET into mvars or savars. You must RGET the field into a temporary avar or svar, and then transfer into the subscripted variable.

The following example RGETs 20 records of the form "Name", "Address", "City", "State", Zip, Phone, and stores them in string and arithmetic arrays, dependent upon the data type of the field:

```
100 Dim Names$(20,30),Addrs$(20,30),Cities$(20,20),States$(20,2)
110 Dim Tname$(30),Taddr$(30),Tcity$(20),Tstate$(2)
120 Dim Zips(20),Phones(20)
130 Close #1:Open #1,4,0,"D:FRIENDS.DAT"
140 For Recnum=1 To 20
150   Rget #1,Tname$,Taddr$,Tcity$,Tstate$,Tzip,Tphone
160   Names$(Recnum;)=Tname$:Addrs$(Recnum;)=Taddr$:Cities$(Recnum;)=Tcity$
170   States$(Recnum;)=Tstate$:Zips(Recnum)=Tzip:Phones(Recnum)=Tphone
180 Next Recnum
190 Close #1:Print :Print "Got File"
200 Rem "Now that we have records, let's show them"
210 Input "Record to View? ",Recnum
220 If Recnum<>0:If Recnum>20 Then 300
230   Gosub 310
240 Else :Rem "show all records"
250   For Recnum=1 To 20
260     Gosub 310
270   Next Recnum
280 Endif
290 Goto 210
300 End
310 Print Names$(Recnum;):Print Addrs$(Recnum;)
320 Print Cities$(Recnum;)", ";States$(Recnum;)" ";Zips(Recnum)
330 Print  Using "(###)###/-####",Phones(Recnum):Print
340 Return
```

BSAVE

Format: BSAVE aexp1,aexp2,"filespec"
Example: BSAVE $680,$6FF,"D:PAGEFLIP.BIN"

BSAVE allows you to store a binary image in standard Atari DOS LOAD format (with header) so that you can later **BLOAD** it directly into the right place. **aexp1** is the starting address of the region of memory you want to save, and **aexp2** is the ending address of the region. A total of aexp2-aexp1+1 bytes of binary data are stored.

Technical Note: BSAVE saves the memory image as a single segment, with a single header. No RUN or INIT vector is appended.

BLOAD

Format: BLOAD "filespec"
Example: BLOAD "D:PAGEFLIP.BIN"

BLOAD is the complementary statement to **BSAVE** because it allows you to load a standard Atari DOS LOAD format binary file. It can also be used to load USR routines you have written using MAC/65 (or some other inferior assembler).

Warning: BLOAD performs no checks of the addresses specified in the segment header(s). You can easily wipe out huge and important parts of memory with this statement!

Technical Note: BLOAD will load binary files that are made up of any number of segments. It will load but ignore RUN and/or INIT vectors.

Bonus: if your binary file has a RUN vector, you can execute it via **SET 8,0:A=USR(DPEEK($2E0))**.

NOTE (NO.)

Format: NOTE #chan, avar1, avar2
Example: 100 NOTE #1,X,Y

NOTE stores the current disk sector number in avar1 and the current byte offset within that sector in avar2. This is the current read or write position in the specified file where the next byte to be read or written is located.

POINT (P.)

Format: POINT #chan, avar1, avar2
Example: 100 POINT #2, A, B

POINT sets the current disk sector to avar1, and the current byte number within that sector to avar2. Essentially, it moves a software-controlled pointer to the specified location in the file. This gives the user "random" access to the data stored on a disk file. The **POINT** and **NOTE** commands are discussed in more detail in your DOS Manual.

STATUS (ST.)

Format: STATUS #chan, avar
Example: 350 STATUS #1,Z

STATUS calls the status routine for the device OPENed on channel chan, and stores the value returned in avar. This can be useful when dealing with devices that produce special status values (e.g., R:).

Warning: if no device is currently **OPEN** on chan, **STATUS** will still try to do something. What it will do depends on the last thing that was done on channel chan, and can produce disastrous results. We strongly recommend using XIO 13 on channels which are not OPEN.

XIO (X.)

Format: XIO cmdno, #chan, aexp1, aexp2, "filespec"
Example: XIO 18,#6, 0, 0, "S:"

XIO is a general input/output statement that allows you to access the special capabilities of the device filespec. cmdno is an aexp, and specifies the function you wish the device to perform. aexp1 and aexp2 are put in the aux1 and aux2 bytes of channel chan, and are dependent upon the function. A list of useful cmdnos follows:

cmdno	operation	example
3	Open	Use OPEN instead
5	Get Text	Use INPUT instead
7	Get Char	Use GET or BGET instead
9	Put Text	Use PRINT instead
11	Put Char	Use PUT or BPUT instead
12	Close	Use CLOSE instead
13	Status	XIO 13,#6,0,0,"R4:"
17	Draw Line	Use DRAWTO instead
18	Fill	XIO 18,#6,0,0,"S:"
32	Rename File	Use RENAME instead
33	Delete File	Use ERASE instead
35	Lock File	Use PROTECT instead
36	Unlock File	Use UNPROTECT instead
37	Disk Point	Use POINT instead
38	Disk Note	Use NOTE instead
253	2.5 Format	XIO 253,#1,$22,0,"D2:"
254	Disk Format	XIO 254,#1,0,0,"D2:"

Note: we strongly recommend that you use only cmdno's 13, 18, 253, and 254, since BASIC XE has statements that perform all the others.

Managing Disk Files

The statements in this chapter allow you to perform DOS-type commands without ever leaving BASIC XE. The statements are **DIR**, **PROTECT**, **UNPROTECT**, **RENAME**, and **ERASE**.

Note: In the examples in this chapter, you will sometimes see the wildcard characters * and ? in the filespec. For information on the use of these, see your DOS manual.

DIR

```
Format:    DIR ["filespec"]
Examples:  100 DIR "D:*.COM"
           DIR FILE$
           DIR "D2:TEST?.B*"
```

The **DIR** command shows a list of the disk files which match filespec, and is similar to the DOS XL DIR command. If no filespec is given all files on D1: are displayed. The first example will display all files on D1: with the "COM" extension. The second example shows a string variable being used as filespec. This is legal, but the string variable must contain a valid filespec, otherwise an error will occur. The third example will display all files on the disk in drive 2 which match TEST?.B*.

Note: **DIR** must be used as the last (or only) command on a progam line.

PROTECT

```
Format:    PROTECT "filespec"
Examples:  PROTECT "D:*.COM"
           100 PROTECT "D2:FILE.BXE"
```

PROTECT allows you to protect your disk files without going to DOS, and is very similar to the DOS XL PRO command.

Note: Atari DOS uses the terms 'LOCK' and 'UNLOCK' instead of **PROTECT** and **UNPROTECT**. They're just different names for the same idea.

UNPROTECT (UNP.)

```
Format:    UNPROTECT "filespec"
Examples:  100 UNPROTECT "D:DATA.001"
           UNP. "D2:*.*"
```

The **UNPROTECT** statement allows you to unprotect disk files which have been protected using either the BASIC XE **PROTECT** statement or the DOS XL PRO command, and is similar to the DOS XL command UNProtect.

RENAME

Format: RENAME "filespec,filename"
Example: RENAME "D2:OLDNAME.EXT,NEWNAME.EXT"

RENAME allows you to rename disk files directly from BASIC XE. Note: the comma shown between filespec and filename is required.

Caution: the new filename cannot include a device specifier (Dn:). Also, we strongly suggest that you do not use wildcards when **RENAME**ing.

ERASE

Format: ERASE filespec
Examples: ERASE "D:*.BAK"
ERASE "D2:TEST?.SAV"

ERASE will erase any unprotected files which match the given filespec. The first example above would erase all files on the disk in drive 1 with the extension "BAK". The second example would erase all files matching TEST?.SAV on the disk in drive 2. This command is similar to DOS XL's ERA.

Looping and Jumping Statements

The statements discussed in this chapter allow you to have repetition and iteration in your BASIC XE programs without a lot of trouble. The looping statements are **FOR** and **WHILE**, and the jumping statement is **GOTO**. The **POP** statement is also included because it directly affects the execution of the other three.

FOR / STEP / NEXT

Format: FOR avar=aexp1 TO aexp2 [STEP aexp3]
[statements]
NEXT avar

The **FOR** statement is used to repeat a group of statements a specified number of times. It does this by initializing the loop variable (avar) to the value aexp1. Each time the **NEXT** avar statement is encountered, avar is incremented by aexp3 if the **STEP** option is used. If this option is not used, avar is incremented by 1. When avar becomes greater than aexp2, the loop stops executing, and the program proceeds to the statement immediately following the NEXT avar. You can control whether or not a FOR loop will execute at least once (a la Atari BASIC) using **SET 3,aexp**.

FOR loops can be nested (one **FOR** loop within another). In this case, the innermost loop is completed before returning to the outer loop. The following program is an example of nesting (notice how LIST indents loops to show the statements within a loop):

```
10 For X=1 To 3
20   Print "X Loop: ";X
30   For Y=1 To 5 Step 2
40     Print "  Y Loop: ";Y;
50   Next Y
60   Print
70 Next X
80 End
```

The outer loop will complete three passes (X=1 to 3). However, before this first loop reaches its **NEXT X** statement, the program gives control to the inner loop. Note that the **NEXT** statement for the inner loop must precede the **NEXT** statement for the outer loop. In the example, the inner loop's number of passes is determined by the **STEP** statement (**STEP 2**). Using this data, the computer must complete three passes through the inner loop before the inner loop counter (**Y**) becomes greater than 5. The following is the output of this program when it is **RUN**:

```
X Loop: 1
  Y Loop: 1  Y Loop: 3  Y Loop: 5
X Loop: 2
  Y Loop: 1  Y Loop: 3  Y Loop: 5
X Loop: 3
  Y Loop: 1  Y Loop: 3  Y Loop: 5
```

WHILE / ENDWHILE

Format: WHILE aexp
[statements]
ENDWHILE

WHILE allows you a looping statement which continues execution conditionally. So long as **aexp** is non-zero (it can be either positive or negative), all statements between **WHILE** and **ENDWHILE** will be executed. Before each pass through the statements in the loop, aexp is evaluated to determine whether loop execution should continue or not. For example, **WHILE 1** will execute forever, and **WHILE 0** will never execute. The following program is an example of the WHILE loop:

```
100 Rmax=5:Cmax=8:Currow=0:Curcol=0:Found=0:Target=0
105 Dim Matrix(Rmax,Cmax)
110 While Currow<Rmax And ( Not Found)
120   Curcol=0
130   While Curcol<Cmax And ( Not Found)
140     If Matrix(Currow,Curcol)=Target Then Found=1
150     Curcol=Curcol+1
160   Endwhile
170   Currow=Currow+1
180 Endwhile
190 If Found:Print "Found ";Target;" at ";
200   Print "Matrix(";Currow-1;",";Curcol-1;")"
210 Else :Print Target;" not found"
220 Endif
```

GOTO (G.)

Format: GOTO lineno

The GOTO command is used to jump unconditionally to another part of the program by specifying a target line number (lineno). Because there is no way to return from a GOTO, the statements which follow it will never be executed, unless of course another GOTO jumps back to them. The following example program shows several uses of GOTO:

```
100 Tryagain=110
110 Input "Give me a number from 1 to 9 > ",Lucky
120 If Lucky<1 Then 110
130 If Lucky>9 Then Goto 110
140 If Lucky<>Int(Lucky) Then Goto Tryagain
150 Print :Print
160 Goto 200+Lucky*10
200 Rem *** CHOOSE A WORD ***
210 Lucky$="Fitch":Goto 300
220 Lucky$="Pippin":Goto 300
230 Lucky$="Mandrill":Goto 300
240 Lucky$="Zeitgeist":Goto 300
250 Lucky$="Zloty":Goto 300
260 Lucky$="Freshet":Goto 300
270 Lucky$="Crosier":Goto 300
280 Lucky$="Brougham":Goto 300
290 Lucky$="Abattoir":Goto 300
300 Print " Your lucky crossword puzzle word is:"
310 Tab (35-Len(Lucky$))/2
320 Inverse :Print Lucky$:Normal :Print
330 Goto Tryagain
```

Note: any GOTO statement that jumps to a preceding line may result in an endless loop.

Note: using anything other than a numeric constant for lineno will make renumbering using RENUM difficult. However, readability may be markedly improved.

POP

Format: POP

To understand what **POP** does, we need to take a little journey inside BASIC XE to find out more about how loops work. When BASIC XE sees a **FOR**, **WHILE**, or **GOSUB**, it saves away its current position in the program. That way, when it reaches the **NEXT**, **ENDWHILE**, or **RETURN**, it will know where to go back to. Also, **LOCAL** saves the previous value of an avar when you make it private so that it can later be restored. The place where BASIC XE saves these things is called the program stack, and is really just a list. Putting something on the stack is called 'pushing', and taking something off is called 'popping', hence the command **POP** suggests that it takes something off the stack. This is exactly what it does, and is very useful when you want

1) to jump out of a loop before it has executed its specified number of times,
2) to get out of a subroutine (**GOSUB**) which does not give control back to the main program through the use of a **RETURN**, or
3) to restore the previous values of **LOCAL** avars, thus ending a **LOCAL** region without an **EXIT**.

Warning: if you **POP** too many or too few items off the stack it will cause an error (13, 16, or 28, dependent upon what you left at the top of the stack).

The following examples illustrate these uses of **POP**:

```
10 For I=0 To 9
20   Print I;
30   Local I
40   I=Random(10,99)
50   Print " : ";I;
60   Pop
70   Print " : ";I
80 Next I
90 Rem lines 20 and 30 may be swapped



100 Print "At line 100"
110 Gosub 200
120 Print "At line 120"
130 End
190 Rem ********************************
200 Print "  At line 200"
210 Gosub 300
220 Print "  At line 220"
230 Goto 200
290 Rem ********************************
300 Print "    At line 300"
310 For I=1 To 5
320   Print "      At line 320"
330   If I=3 And Flag Then Pop :Pop :Return
340 Next I
350 Print "    At line 350"
360 Flag=1
370 Return
```

Conditional Statements

The statements discussed in this chapter allow you to execute parts of your program only if the conditions you specify have been met. The conditional statements are IF/THEN, IF/ELSE/ENDIF, and ON.

IF / THEN

Format: IF aexp THEN | lineno |
| statement[:statement...] |

The IF/THEN conditional is used when you want to execute a group of statements only if certain conditions are met. These conditions may be either arithmetic or logical. If the aexp following the IF is true (non-zero), the program executes the THEN part of the statement. If, however, aexp is false (zero), the rest of the statement is ignored and program control passes to the next numbered line. When THEN is followed by a line number (lineno), execution continues at that program line if aexp is true. Note: lineno must be a constant (not an expression).

Several IF/THEN conditionals may be nested on the same line. In the example,
```
100 If X=5 Then R=9:If Y=3 Then Goto 200
```
the statement R=9 will be executed if X=5, while the statement GOTO 200 will be executed only if X=5 and Y=3.

The following program demonstrates the IF/THEN conditional:

```
100 Graphics 0:Print "IF DEMO"
110 Input "Enter Value 1..3>> ",A
120 If A=1 Then Print "One"
130 If A=2 Then Print "Two"
140 If A=3 Then Print "Three"
150 If A<1 Or A>3 Then Print "Invalid Value"
160 Goto 110
170 End
```

IF / ELSE / ENDIF

Format: IF aexp
[statements]
[ELSE
[statements]]
ENDIF

BASIC XE makes available an exceptionally powerful conditional capability via **IF / ELSE / ENDIF.** If the expression **aexp** is true (non-zero) then all the statements between **aexp** and **ELSE** will be executed, while the statements between **ELSE** and **ENDIF** will be skipped. If **aexp** is false (zero), then the statements between aexp and **ELSE** will be skipped, and those between **ELSE** and **ENDIF** will be executed. If **ELSE** is not used, this conditional acts just like a multi-line **IF/THEN** with **IF** and **ENDIF** as delimiters.

Caution: the keyword **THEN** is not part of the syntax of this conditional.

The following program illustrates **IF / ELSE / ENDIF:**

```
100 If 1<2
110   Print "This ";
120   If 2>3
130     Print "computer ";
140     If 3<4
150       Print "is ";
160     Else
170       Print "broken!"
180     Endif
190   Else
200     Print "program ";
210     If 4>5
220       Print "is a ";
230       If 5<6
240         Print "boo-boo"
250       Endif
260     Else
270       Print "works ";
280       If 6>7
290         Print "poorly."
300       Else
310         Print "great!"
320       Endif
330     Endif
340   Endif
350 Else
360   Print "Kablooey!!!!!"
370 Endif
```

ON

Format: ON aexp |GOTO | lineno1[,lineno2...]
|GOSUB|

Note: GOSUB and GOTO may not be abbreviated when used in conjunction with ON.

The ON statement allows conditional jumps and subroutine calls. The condition is determined by aexp. If it is negative, an error results. If it is non-negative, aexp is rounded to the nearest integer, and program control is channelled according to the following table:

value	Control goes to
0	Statement after ON
1	lineno1
2	lineno2
:	:
N	linenoN
>N	Statement after ON

"N" is the last line number in the list of lineno's following the GOTO or GOSUB. When ON/GOSUB is used, control returns to the statement following the ON/GOSUB after the subroutine RETURNs.

The following program demonstrates the ON statement, both with GOTO and GOSUB:

```
100 Graphics 2:Print #6;"basic xe FILE RUNNER"
110 Print #6
120 Print #6;"1 run basic xe file":Print #6
130 Print #6;"2 disk directory":Print #6
140 Print #6;"3 quit"
150 Input "Your Choice? ",Pick
160 On ((Pick>3) Or (Pick=0)) Goto 150
170 If Pick=3 Then Graphics 0:End
180 On Pick Gosub 200,300
190 On Pick Goto 150,100
200 Trap 280
210 Input "File Name? ",F$
220 If Find(F$,":",0)=0:T$="D:",F$
230 Else :T$=F$
240 Endif
250 If Find(T$,".BXE",0)=0 Then T$=T$,".BXE"
260 Print "Running ";T$;"...";:Run T$
270 Return
280 Trap 0:Print " I/O Error #";Err(0)
290 Return
300 Graphics 0:Print "All Files with '.BXE' Extender:"
310 Trap 360
320 Print :Dir "D:*.BXE"
330 Print :Print "Press START for menu"
340 If Peek($d01f)&1 Then 340
350 Return
360 Trap 0
370 If Err(0)<>136 Then Print " I/O Error #";Err(0)
380 Cont
```

Space For Your Notes

Handling Errors

The statements and function in this chapter allow you to detect and resolve run-time errors without causing program execution to halt. Included are the TRAP statement, the ERR function, and a discussion of the error handling applications of CONT and STOP.

TRAP (T.)

Format: TRAP lineno
Example: 100 TRAP 2000

The TRAP statement is used to direct the program to a specified line number if an error is detected. Without a TRAP the program stops executing when an error is encountered and displays an error message on the screen.

TRAP works for any error that may occur after it (the TRAP statement) has been executed, but once an error has been detected and trapped, it is necessary to reset the error trapping with another TRAP statement. This resetting TRAP should be done at the beginning of the error handling routine, to insure that the TRAP is reset after each error.

To find out the error number and the line number on which the error occured, use ERR, as described in the following section.

TRAP may be disabled by executing a TRAP statement with an lineno value of 0 or greater than 32767.

Examples of TRAP may be found in the program on the following page.

f ERR

Format: ERR(aexp)

This function allows you to find out the error number and line on which the error occurred when you are writing your own error trapping routines. Using an aexp of 0 will return the error number of the last run-time error, and an aexp of 1 will return the program line on which the error occured. The results of using other values of aexp are undefined.

Examples of ERR may be found in the program on the following page.

A Program Example Using TRAP and ERR

```
100 Deg
110 Print "Angle  Sine          Cosecant"
120 For I=0 To 180 Step 15
130   Print  Using "###    #.########      ",I,Sin(I),
140   Trap 200
150   Print  Using "######.######",1/Sin(I)
160 Next I
170 End
180 Rem we get to line 200 if
190 Rem Sin(I) is equal to zero!
200 Print "undefined"
210 Goto Err(1)+10
```

Using STOP & CONT in Error Handling

CONT can be very useful in error handling because you need not fool around with line numbers to continue program execution. In the above example, execution continues on the line following the error through the use of **ERR(1)** and a **GOTO**. If **CONT** is used instead, line 210 becomes much simpler:

```
210 Cont
```

The use of **STOP** in error handling is limited but very useful. In fact, it is not error handling at all; it is error creation. When you are developing a program, you can put **STOP**s where the program should never see them. If you get a "Stopped at lineno", then you know you're doing something wrong.

Handling Strings

This chapter discusses the functions in BASIC XE that are designed to make manipulating string data quick and easy.

f ASC

Format: ASC(sexp)
Example: 100 A=ASC(A$)

ASC returns the ATASCII numeric value of the first character in sexp. If A$= "ABC", then ASC(A$) returns 65, and ASC(A$(2)) returns 66.

Note: Appendix A contains a table of ATASCII codes and characters.

f CHR$

Format: CHR$(aexp)
Examples: PRINT CHR$(65)
100 A$=CHR$(65)

CHR$ returns the character (in string format) represented by the ATASCII numeric code aexp. Only one character is returned. In the above examples, the letter A is returned. Using the ASC and CHR$ functions, the following program prints the upper case and lower case letters of the alphabet:

```
10 For C=0 To 25
20   Print Chr$(Asc("A")+C),Chr$(Asc("a")+C)
30 Next C
```

Note: there may be only one STR$ or CHR$ in a logical comparison because BASIC XE uses a single buffer to create the temporary string which both of these functions use (e.g., IF CHR$(A)=CHR$(B)... is always true, whether A and B are equal or not.

f LEN

Format: LEN(sexp)

The LEN function returns the character length of sexp. This information may then be printed or used later in a program. The length of a string variable is simply the element number of the last character currently in the string. Strings have a length of 0 until characters have been stored in them.

f FIND

Format: FIND(sexp1, sexp2, aexp)
Example: PRINT FIND("ABCDXXXABC", "BC",N)

FIND is an efficient, speedy way of determining whether any given substring is in any given master string. **FIND** will search **sexp1**, starting at position **aexp+1**, for the substring **sexp2**. If **sexp2** is found, the function returns the position where it was found, relative to the beginning of **sexp1**. If **sexp2** is not found, a 0 is returned.

In the example above, the following values would be PRINTed:

2 if N = 0 or 1
9 if N>=2 and N<9
0 if N>=9

The following example shows an easy way to have a vector dependent upon a menu choice:

```
10 Input "[C]hange, [E]rase, or [L]ist? ",A$
20 On Find("CEL",A$(1,1),0) Goto 100,200,300
30 Goto 10
```

This example illustrates how changes to aexp can affect the results of FIND:

```
10 Input "A string, please - ",A$
20 For St=0 To Len(A$)-2
30   F=Find(A$,"A",St)+1
40   If F=1 Then Print "Neither 'AB' nor 'AC' were found":End
50   If A$(F,F)="B" Then Print "Found 'AB' at pos. #";F-1:St=St+1
60   If A$(F,F)="C" Then Print "Found 'AC' at pos. #";F-1:St=St+1
70 Next St
```

f ADR

Format: ADR(sexp)
Examples: ADR(A$)
ADR(B$(5;))

ADR returns the memory address of the string **sexp**. Knowing the address enables you to use it in USR routines, BGET, BPUT, etc.

Warning: if you are in EXTENDed mode, ADR("string") returns an improper value because the string constant is copied out of the banked program memory into a temporary area. Because it's within a single statement,

```
J=Usr(Adr("M.L. in char string"))
```

works, but

```
T=Adr("M.L. in char string"):J=Usr(T)
```

won't because it's two statements. If you use ADR("string") as in the first case only, you can SET 15,1 so that BASIC XE won't force an error.

f LEFT$

```
Format:    LEFT$(sexp, aexp)
Examples:  10 A$=LEFT$("ABCDE",3)
           20 PRINT LEFT$("ABCD",5)
```

The **LEFT$** function returns the leftmost aexp characters of the string sexp. If aexp is greater than the number of characters in sexp, no error occurs and the entire string sexp is returned.

In the first example, A$ is equated to "ABC", and in the second example, the entire string "ABCD" is printed.

f MID$

```
Format:    MID$(sexp,aexp1,aexp2)
Example:   A$=MID$("ABCDEFG",2,4)
```

**MID$** allows you to get a substring from the middle of another string. The substring retrieved starts at the aexp1th character of sexp, and is aexp2 characters long. If aexp1 equals 0 an error occurs (since there is no 0th character in a string); if aexp1 is greater than the LEN length of sexp, no error occurs (and no characters are returned). aexp2 may be any positive integer, but if its value makes the substring go beyond the LEN length of sexp, then the substring returned ends at the end of sexp.

In the above example, A$ is equated to "BCDE".

f RIGHT$

```
Format:    RIGHT$(sexp,aexp)
Example:   A$=RIGHT$("123456",4)
```

The **RIGHT$** function returns the rightmost aexp characters of sexp. If aexp is greater than the number of characters in sexp, then the entire string sexp is returned.

In the above example, A$ is equated to "3456".

f VAL

Format: VAL(sexp)
Example: 100 A=VAL(A$)

VAL returns the numeric value represented by a string, providing that the string is indeed a string representation of a number (i.e. is a digit string). Using this function, the computer can perform arithmetic operations on strings as shown in the following example program:

```
10 B$="10000"
20 B=Sqr(Val(B$))
30 Print "The Square Root of ";B$;" is ";B
```

Note: VAL does not permit the use of an sexp that does not start with a digit (i.e., that cannot be interpreted as a number). It can, however, interpret floating point numbers (e.g., VAL("1E5") would return the number 100,000). Also, non-numeric characters following a valid digit string will be ignored (e.g., VAL("100ABC") returns 100).

Note: VAL will convert hex digit strings if they begin with a "$". (You can disallow this via SET 13,0).

f STR$

Format: STR(aexp)
Example: A$=STR$(650)

STR$ returns the string form of aexp. The above example would return the actual number 650, but as the string "650".

Warning: may be only one STR$ or only one CHR$ in a logical comparison. See CHR$ for more info.

f HEX$

Format: HEX$(aexp)
Examples: PRINT HEX$(5000)
PRINT "$";RIGHT$(HEX$(32),2)

The HEX$ function will convert aexp to a four digit hexadecimal number in string format (the second example shows how to get a two digit hex number).

Note: no dollar sign ($) is placed in front of the hex digit string.

Using the Game Controllers

The functions discussed in this chapter allow you to access the paddle, joystick, and light pen easily and quickly.

f PADDLE

Format: PADDLE(aexp)
Example: PRINT PADDLE(3)

The **PADDLE** function returns the current value of the paddle in port aexp (0-3). The value returned will be between 1 and 228, inclusive, with the value increasing as the paddle knob is turned counterclockwise.

f PTRIG

Format: PTRIG(aexp)
Example: 100 IF PTRIG(1)=0 THEN PRINT "Missile Fired!"

PTRIG returns a 0 if the trigger button of the paddle in port aexp (0-3) is pressed. Otherwise, it returns a value of 1.

f PEN

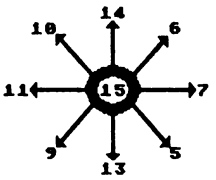
Format: PEN(aexp)
Example: PRINT "light pen at ";PEN(0);",";PEN(1)

The **PEN** function simply reads the ATARI light pen registers and returns their contents. If aexp is 0, the horizontal position is returned; if aexp is 1, the vertical position is returned.

f STICK

Format: STICK(aexp)
Example: 100 PRINT STICK(1)

The **STICK** function returns the position value of the joystick in port aexp (0-1), as defined in the following diagram:



f HSTICK

Format: HSTICK(aexp)

The HSTICK function returns an easily usable code for horizontal movement of a given joystick. aexp is simply the number of the joystick port (0-1), and the values returned (and their meanings) are as follows:

```
-1 if the joystick is pushed left
 0 if the joystick is centered
+1 if the joystick is pushed right
```

Here is an example of HSTICK in use:

```
10 Let Dir=Hstick(0)
20 If Dir=-1 Then Print "↑ Left"
30 If Dir=0 Then Print "  Stopped"
40 If Dir=1 Then Print "↑ Right"
50 Goto 10
```

f VSTICK

Format: VSTICK(aexp)

The VSTICK function returns an easily usable code for vertical movement of a given joystick. aexp is simply the number of the joystick port (0-1), and the values returned (and their meanings) are as follows:

```
-1 if the joystick is pushed down
 0 if the joystick is centered
+1 if the joystick is pushed up
```

Here is an example of VSTICK in use:

```
10 Let Dir=Vstick(0)
20 If Dir=-1 Then Print "↓ Down"
30 If Dir=0 Then Print "  Stopped"
40 If Dir=1 Then Print "↑ Up"
50 Goto 10
```

f STRIG

Format: STRIG(aexp)
Example: 100 IF STRIG(1)=0 THEN PRINT "Fire Torpedo"

The STRIG function works the same as the PTRIG function, except that it is used with the joysticks instead of the paddles. aexp specifies the joystick port (0-1).

Graphics

This chapter describes the BASIC XE statements that allow you to manipulate the wide variety of screen graphics available on the Atari personal computers. Before going into the graphics commands, a little background about the modes available would be useful.

Introducing Atari Graphics

The table below summarizes the graphics modes available via BASIC XE. A quick glance down the "Type" column will show you that the Atari supports two types of graphics, text and grid. In text graphics each pixel represents an ATASCII character, while in the grid modes a pixel represents a box of color. The size of a pixel depends upon the graphics mode. In all graphics modes, position 0,0 is at the upper left corner of the graphics area; moving right increases the column value, and moving down increases the row value. The diagram at the end of this section illustrates this coordinate system visually.

If you look at the column headings in the table, you will notice two "Rows" columns. "Split Rows" is the number of rows when you are using the graphics mode in conjunction with a text window, and "Full Rows" refers to the number of rows when used without the text window.

Following the table are short descriptions of these graphics modes.

Mode	Type	Columns	Split Rows	Full Rows	Colors
0	Text	40	N/A	24	1.5
1	Text	20	20	24	5
2	Text	20	10	12	5
3	Grid	40	20	24	4
4	Grid	80	40	48	2
5	Grid	80	40	48	4
6	Grid	160	80	96	2
7	Grid	160	80	96	4
8	Grid	320	160	192	1.5
9	Grid	80	N/A	192	16
10	Grid	80	N/A	192	9
11	Grid	80	N/A	192	16
12	Text	40	20	24	4-5
13	Text	40	10	12	4-5
14	Grid	160	160	192	2
15	Grid	160	160	192	4

Mode 0: this mode is the 1 color, 2 luminance (brightness) default mode for Atari Personal Computers. It contains a 24 line by 40 character screen matrix. The default margin settings of 2 and 39 allow 38 characters per line. Margins may be changed by POKEing LMARGN and RMARGN (82 and 83). Some systems have different margin default settings. The color of the characters is determined by the background color. Only the luminance of the characters can be different.

Modes 1 and 2: these two 5-color modes are text modes. Characters in mode 1 are twice the width of those in mode 0, but are the same height, while those in mode 2 are twice the width and twice the height of those in mode 0. In the split-screen mode, **PRINT** will print data in the text window, and **PRINT #6** will print data in the mode 1 or 2 graphics window.

The default colors depend on the type of character input, as defined in the following table:

Character Type	SETCOLOR Register	Default Color
0..9 & A..Z	0	Orange
Cntl Chrs & a..z	1	Light Green
Inverse 0..9 & A..Z	2	Dark Blue
Inverse Cntl Chrs & a..z	3	Red
Playfield and Border	4	Black

Note: see **SETCOLOR** to change character colors.

Unless otherwise specified, all characters are displayed in uppercase non-inverse form. To print lowercase letters and graphics characters, use a **POKE $2F4,$E2.** To return to upper case, use **POKE $2F4,$E0.**

Modes 3, 5, 7, and 15: these four 4 color grid modes are also split-screen displays in their default state, but may be changed to full screen by adding 16 to the mode number. Modes 3, 5, and 7 differ only in grid size. In mode 15 the pixels are smallest, thereby giving the highest resolution.

Modes 4, 6, and 14: these three 2-color grid modes have an advantage over the 4-color grid modes in that they require less RAM space. Therefore, they may be used when only two colors are needed and RAM is getting crowded.

Mode 8: this grid mode gives the highest resolution of all. As it takes a lot of RAM to obtain this kind of resolution, it can only accommodate a maximum of one color and two different luminances, as mode 0.

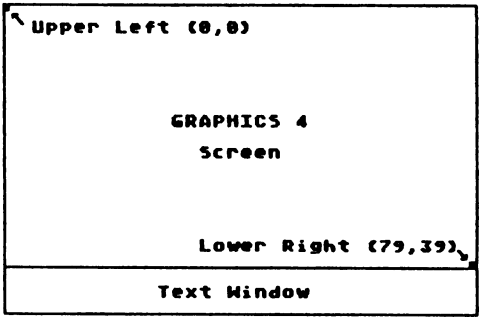
Modes 12 and 13: these two text modes are very special. Instead of using single bits within a characters definition in the character set to determine how to represent that character, they use bit pairs and interpret them as colors, as follows:

Bit Image	SETCOLOR Register
00	4
01	0
10	1
11	2 / 3*

* If the character is in inverse video, register 3 is used, otherwise register 2 is used. This enables you to have 5 color on the screen at one time, although you may have only 4 colors in a single character.

Modes 9, 10, and 11: these are the GTIA modes, and are somewhat different from all the other modes. Note that these modes do not allow a text window. **Mode 9** is a one color, 16 luminance mode. The main color is set by the background color, and the luminance values are determined by the information in the screen memory itself. Each pixel is four bits wide, allowing for 16 different values (0-15). These values are interpreted as the luminance of the base color for that pixel. **Mode 11** is similar to mode 9 in that the color information is in the screen memory itself, but the information for each pixel is interpreted as a color instead of a luminance. Thus there are 16 colors, all of the same luminance. The luminance is set by the luminance of the background color (default is 6). **Mode 10** is somewhat of a crossbreed of the other two GTIA modes and the normal modes in that it offers lots of colors (like the GTIA modes) and uses the color registers (like the normal modes). However, since mode 10 allows 9 colors, it must use the player color registers as well as the other color registers. The following table shows how the pixel values relate to the color registers and what BASIC XE command may be used to set each color register.

Pixel Value	System Register	Reg. Addr	BASIC XE Statement
0	PCOLR0	704	PMCOLOR 0,aexp
1	PCOLR1	705	PMCOLOR 1,aexp
2	PCOLR2	706	PMCOLOR 2,aexp
3	PCOLR3	707	PMCOLOR 3,aexp
4	COLOR0	708	SETCOLOR 0,aexp
5	COLOR1	709	SETCOLOR 1,aexp
6	COLOR2	710	SETCOLOR 2,aexp
7	COLOR3	711	SETCOLOR 3,aexp
8	COLOR4	712	SETCOLOR 4,aexp



GRAPHICS (GR.)

Format: GRAPHICS aexp
Example: GRAPHICS 2

The **GRAPHICS** statement is used to select one of the graphics modes discussed above. It automatically opens the graphics area of the screen (S:) on channel #6. As a result of this, it is not necessary to specify a channel number when you want to PRINT to the text window, since it is still open on channel #0. aexp is the mode number as used in the table at the start of this chapter, and must be positive.

Modes 0, 9, 10, and 11 are full-screen display only, while modes 1 through 8 are default to split-screen displays. To override the split-screen, add 16 to the mode number (aexp). Adding 32 prevents GRAPHICS from clearing the screen memory.

SETCOLOR (SE.)

Format: SETCOLOR aexp1,aexp2,aexp3
Example: 100 SETCOLOR 0,1,4

SETCOLOR is used to set the hue and luminance of one of the color registers. aexp1 is the number of the color register (values 0-4 legal), aexp2 is the hue (see following table), and aexp3 is the luminance (0-14, even numbers only, are valid). the larger aexp3 is, the brighter the color. The following table shows the aexp2 values and corresponding colors:

aexp2	Color	aexp2	Color
0	Gray	8	Blue
1	Gold	9	Light Blue
2	Orange	10	Turquoise
3	Red-Orange	11	Green-Blue
4	Pink	12	Green
5	Violet	13	Yellow-Green
6	Blue-Violet	14	Orange-Green
7	Blue	15	Light Orange

Note: actual colors will vary with type and adjustment of TV or monitor used.

The following table shows the default values for the five **SETCOLOR** registers:

Reg	Value	Color	Lum	Color
0	$28	2	8	Orange
1	$CA	12	10	Green
2	$94	9	4	Dark Blue
3	$46	4	6	Pink-Red
4	$00	0	0	Black

SETCOLOR uses values 0 to 4 to specify the color register, while **COLOR** uses different values. Translation between the two can be confusing, so careful study of the table on the following page is advised.

SETCOLOR / COLOR Table

GR Mode	COLOR value	SE. reg	Description and Comments
0 and all text windows	COLOR value picks chr to PLOT, DRAW, etc	1	Character Luminance
		2	PF Color & Char Hue
		4	Border Color
1,2		0	0..9, A..Z
		1	a..z, CNTL A..Z
		2	0..9, A..Z
		3	0..9, CNTL A..Z
		4	PF and Border
3, 5, 7, 15	1	0	Pixel
	2	1	Pixel
	3	2	Pixel
	0	4	Pixel, PF, & Border
4,6,14	1	0	Pixel
	0	4	Pixel, PF, & Border
8	1	1	Pixel Luminance
	0	2	PF Color, Pixel Hue
		4	Border Color
9	0..15 picks Pixel Lum	4	PF & Border Color, Hue of all Pixels NOTE: Reg4 Lum ORed with Pixel Lum to get final Lum.

GR Mode	COLOR value	SE. reg	Description and Comments
10	0	PM0	PF and Border
	1	PM1	Pixel
	2	PM2	Pixel
	3	PM3	Pixel
	4,12	0	Pixel
	5,13	1	Pixel
	6,14	2	Pixel
	7,15	3	Pixel
	8..11	4	Pixel
11	0..15 picks Pixel Hue 0..15	4	PF & Border Color, Lum of all Pixels. NOTE: Reg4 Hue ORed with Pixel Hue to get final Hue.
12,13	COLOR value picks chr to PLOT, DRAW, etc	0	Bit Pair 01
		1	Bit Pair 10
		2	Bit Pair 11, if chr is NORMAL video.
		3	Bit Pair 11, if chr is INVERSE video.
		4	Bit Pair 00

COLOR (C.)

Format: COLOR aexp
Examples: 110 COLOR ASC("A")
COLOR 3

The COLOR statement lets you choose which color will be used for all subsequent PLOTs and DRAWTOs. The aexp value chooses the color and so must be a positive integer 0..255. The color you get is dependent upon the graphics mode you're in, as described in the table above.

Note: in text modes 0, 1, and 2, the number can be from 0 through 255 (8 bits) and determines the character to be displayed (and its color in modes 1 & 2).

Note: when BASIC XE is first powered up COLOR 0 is the default.

PLOT (PL.)

Format: PLOT aexp1,aexp2
Example: 100 PLOT 5,5

The **PLOT** command is used to plot a pixel in the graphics window. **aexp1** specifies the column (X-coordinate) of the pixel, and **aexp2** specifies the row (Y-coordinate). The color of the plotted point is determined by the last **COLOR** statement executed. To change this color (and the color of the PLOTted point) use **SETCOLOR**. Valid pixel coordinates are dependent on the graphics mode being used. The range of points begins at (0,0), and extends to (columns in mode)-1 in the x direction, and (rows in mode)-1 in the y direction.

DRAWTO (DR.)

Format: DRAWTO aexp1,aexp2
Example: 100 DRAWTO 10,8

The **DRAWTO** statement draws a line from the current position of the graphics cursor (set by a previous **PLOT, POSITION,** or **DRAWTO**) to the location **(aexp1,aexp2)**. **aexp1** represents the X coordinate (column) and **aexp2** represents the Y-coordinate (row). The color of the line is determined by the last **COLOR** statement.

POSITION (POS.)

Format: POSITION aexp1,aexp2
Example: 100 POSITION 0,0

POSITION places the invisible graphics cursor at the location **(aexp1,aexp2)** on the screen, and may be used in all graphics modes. In mode 0 only, **POSITION** affects the text cursor, not the graphics cursor.

Note: the cursor does not actually move until the next command that uses the cursor.

LOCATE (LOC.)

Format: LOCATE aexp1,aexp2,avar
Example: 150 LOCATE 11,15,X

The **LOCATE** statement retrieves the value of the pixel at coordinates **(aexp1,aexp2)**, and stores it in avar.

XIO (X.) Fill

Format: XIO 18,#6,0,0,"S:"

This special application of the XIO statement fills an area on the screen between previously PLOTted and DRAWTOed bounds with a non-zero COLOR value. The zeroes in the XIO are used as dummies, but are required. The following steps illustrate the fill process:

1. Pick the COLOR.
2. PLOT bottom right corner.
3. DRAWTO upper right corner.
4. DRAWTO upper left corner.
5. POSITION the cursor at the lower left corner.
6. POKE address 765 with the fill COLOR value.
7. Make the XIO Fill call.

This method is used to fill each horizontal line from top to bottom of the specified area. The fill starts at the left and proceeds across the line to the right until it reaches a pixel which contains non-zero data (will wraparound if necessary). This means that XIO Fill cannot be used to change an area which has been filled in with a non-zero value, as the fill will stop.

Warning: XIO Fill will go into an infinite loop if you attempt to put COLOR 0 on a line which has no non-zero pixels. Pressing <BREAK> or <SYSTEM RESET> can be used to stop the fill if this happens.

Space For Your Notes

Player/Missile Graphics

This chapter describes the BASIC XE commands and functions used to access the Atari's Player-Missile Graphics. Player Missile Graphics (hereafter usually referred to as simply "PMG") represent a portion of the Atari hardware totally ignored by Atari BASIC and Atari OS. Even the screen handler (the S: device) knows nothing about PMG.

BASIC XE goes a long way toward remedying these omissions by adding seven PMG statements and two PMG functions to the already comprehensive Atari graphics. In addition, four other statements and two functions have significant uses in PMG and will be discussed in this chapter.

Introducing P/M Graphics

For a complete technical discussion of PMG, and to learn of even more PMG "tricks" than are included in BASIC XE, read the Atari document entitled "Atari 400/800 Hardware Manual" (Atari part number C016555, Rev. 1 or later).

We stated above that the S: device driver knows nothing of PMG, and in a sense this is proper: the hardware mechanisms that implement PMG are, for virtually all purposes, completely separate and distinct from the "playfield" graphics supported by S:. For example, the size, position, and color of players on the video screen are completely independent of the **GRAPHICS** mode currently active. In Atari (and now BASIC XE) parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical stripe on the screen. Sounds dull? Consider: each player (there are four) may be "painted" in any of the 128 colors available on the Atari (see **SETCOLOR** for specific colors). Within the vertical stripe, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to 0 paints no color at all! That is, any 0 bit in a player stripe has no effect on the underlying playfield display.

Why a vertical stripe? Refer to the figure at the end of this section for a rough idea of the player concept. If we define a shape within the bounds of this stripe (by changing some of the player's bits to 1's), we may then move the stripe anywhere horizontally by a simple register **POKE** (or via the **PMMOVE** statement in BASIC XE). We may move the player vertically by doing a simple circular shift on the contiguous memory block representing the player (again, the **PMMOVE** statement simplifies this process).

To simplify:

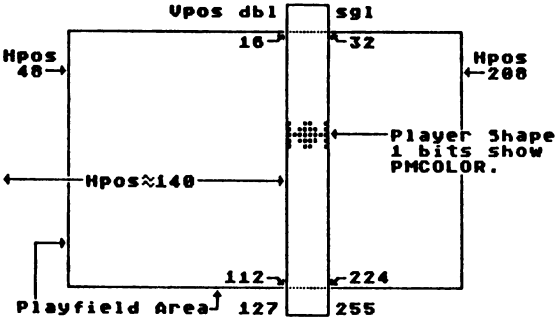
> A player is actually seen as a stripe on the screen 8 pixels wide by 128 (or 256, see below) pixels high. Within this stripe, you can **POKE** or **MOVE** bytes to establish what is essentially a tall, skinny picture (though much of the picture may consist of 0 bits, in which case the background "shows through"). Using **PMMOVE**, you may then move this player to any horizontal or vertical location on the screen.

To complicate:

For each of the four players there is a corresponding "missile" available. Missiles are exactly like players except that:

1) they are only 2 bits wide, and all four missile share a single block of memory.
2) each 2 bit sub-stripe has an independent horizontal position.
3) a missile always has the same color as its parent player.

Again, by using the BASIC XE statements (MISSILE and PMMOVE, for example), you the programmer need not be too aware of the mechanisms of PMG.



P/M Graphics Conventions

1. Players are numbered from 0 through 3. Each player has a corresponding missile whose number is 4 greater then that of its parent player, thus missiles are numbered 4 through 7. In the BUMP function, the "playfields" are actually the colors as defined by SETCOLOR, but are 8 grater than the SETCOLOR register value, and so are numbered 8 - 11.

2. There is some inconsistency in which way is "up". PLOT, DRAWTO, etc. are aware that 0,0 is the top left of the screen and that vertical position numbering increases as you go down the screen. PMMOVE and VSTICK, however, do only relative screen positioning, and define "+" to be up and "-" to be down.

3. "pmnum" is an abbreviation for Player-Missile Number and must be a number from 0 to 3 (for players) or 4 to 7 (for missiles).

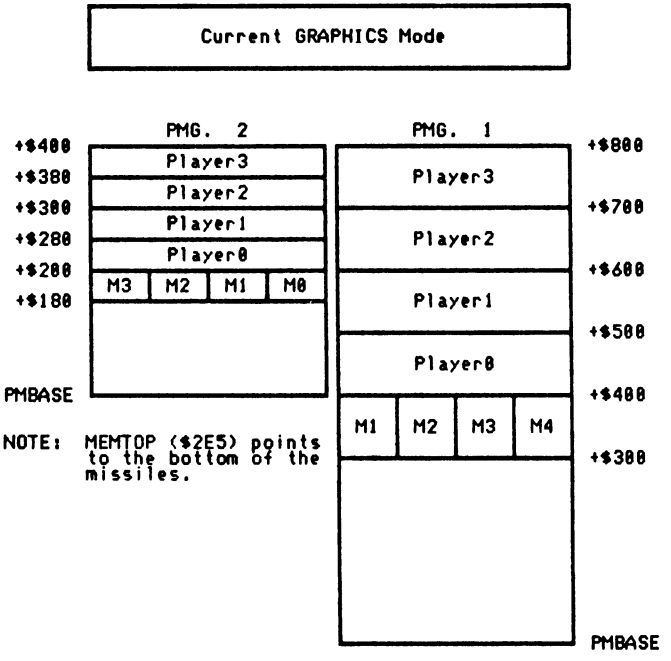
PMGRAPHICS (PMG.)

Format: PMGRAPHICS aexp
Example: PMG. 2

This statement is used to enable or disable the Player/Missile Graphics system. aexp should evaluate to 0,1, or 2, as follows:

0 - Turn off PMG
1 - Enable PMG, single line resolution
2 - Enable PMG, double line resolution

Single and Double line resolution (hereafter refered to as "PMG Modes") refer to the height which a byte in the player "stripe" occupies - either one or two television scan lines (**GRAPHICS 7** has pixels 2 scan lines high, like **PMG. 2**, and **GRAPHICS 15** has pixels 1 scan line high, like **PMG. 1**). The secondary implication of single line versus double line resolution is that single line resolution requires twice as much memory space as double line - 256 bytes per player versus 128 bytes. The following diagram shows PMG memory usage in BASIC XE, but you really need not be aware of the mechanics if you use the **PMADR** function:



PMCOLOR (PMCO.)

Format: PMCOLOR pmnum,aexp1,aexp2
Example: PMCOLOR 2,12,8

PMCOLOR is identical to **SETCOLOR** in usage except that a P/M color register rather than a playfield graphics color register is set to hue **aexp1** and luminance **aexp2**. Note: there is no correspondence in PMG to the **COLOR** statement of playfield graphics - none is necessary since each player has its own color.

The example above would set player 2 and missile 6 to a medium (luminance 8) green (hue 12).

Note: PMG has no default colors set on power-up or <SYSTEM RESET>.

PMMOVE

Format: PMMOVE pmnum [,aexp1] [;aexp2]
Examples: PMMOVE 0,120;1
PMMOVE 1,80
PMMOVE 4;-3

Once a player or missile has been "defined" (via **POKE, MOVE, GET, BGET,** or **MISSILE**), the truly unique features of PMG under BASIC XE may be utilized. With PMMOVE, you may position each P/M shape anywhere on the screen independently in the blink of an eye. Because of the hardware implementation, though, there is a difference in how horizontal and vertical positions are specified.

aexp1 is taken to be the absolute position of the left edge of the "stripe" to be displayed. This position ranges from 0 to 255, though the lowest and highest positions in this range are beyond the edges of the display screen. Note: changing a player's width (see PMWIDTH) will not change the position of its left edge, but will expand the player to the right.

aexp2 is a relative vertical movement specifier. Recall that a "stripe" of player is 128 or 256 bytes of memory. Vertical movement must be accomplished by actual movement of the bytes within the stripe - towards either higher memory (down the screen) or lower memory (up the screen). BASIC XE allows you to specify a vertical movement between -255 (down 255 pixels) and +255 (up 255 pixels), inclusive.

Note: the +/- convention on vertical movement conforms to the value returned by VSTICK. For example, **PMMOVE 2;VSTICK(2)** will move player 2 up or down (or not move him) in accordance with the joystick position.

Note: **SET 7,aexp** may be used to tell **PMMOVE** whether a P/M should "wrap around" (from bottom of screen to top of screen or vice versa) or should disappear as it scrolls off the screen.

MISSILE (MIS.)

Format: MISSILE pmnum,aexp1,aexp2
Example: MISSILE 4,48,3

The **MISSILE** statement allows an easy way for a parent player to "shoot" a missile. **pmnum** is the missile number (4-7), **aexp1** specifies the absolute vertical position of the beginning of the missile (0 is the top of missile memory), and **aexp2** specifies the vertical height of the missile. For example, **MISSILE 4,64,3** would place a missile 3 PMG pixels high at pixel 64 from the top.

Note: **MISSILE** does not simply turn on the bits corresponding to the position specified. Instead, the bits specified are exclusive-or'ed with the current missile memory. This allows you to erase the previous missile **pmnum** when creating another. For example:

```
10 Missile 4,40,1
20 Missile 4,41,1
```

The first statement creates a missile 1 PMG pixel high at vertical position 40. The second statement erases the first missile while creating another 1 PMG pixel missile at vertical position 41, thus giving the effect of a moving missile.

PMWIDTH (PMW.)

Format: PMWIDTH pmnum,aexp
Example: PMWIDTH 1,2

Just as **PMGRAPHICs** allows you to select single or double pixel height, **PMWIDTH** allows you to specify the screen width of players and missiles. However, where **PMGRAPHICs** selects the vertical resolution mode for all players and missiles, **PMWIDTH** allows the width of each player or missile to be specified separately . **aexp** is used for the width and should have a value of 1, 2, or 4 - representing the number of color clocks (equivalent to a pixel width in GR. 7) wide each bit in a player definition will be.

Note: **PMG. 2** and **PMWIDTH 1** combine to allow each bit of a player definition to be equivalent in size to a **GR. 7** pixel, while **PMG. 1** and **PMWIDTH 1** combine to be equivalent to a **GR. 15** pixel - not altogether accidental occurences.

Note: although players may be made wider with **PMWIDTH**, the resolution then suffers. Wider high-resolution "players" may be made by placing two or more separate players side-by-side (as in the second example program at the end of this chapter).

PMCLR (PMC.)

Format: PMCLR pmnum
Example: PMCLR 4

PMCLR "clears" a player or missile area to all zero bytes, thus "erasing" the P/M. **PMCLR** is aware of what PMG mode is active and clears only the appropriate amount of memory. Caution: pmnum values 4 through 7 all produce the same action - all missiles are cleared, not just the one specified. To clear a single missile, try **SET 7,0 : PMMOVE N;255.**

f BUMP

Format: BUMP(pmnum,aexp)
Example: IF BUMP(4,1) THEN B=BUMP(0,8)

BUMP accesses the P/M collision registers of the Atari and returns a 1 (collision occurred) or 0 (no collision occurred) as appropriate for the pair of objects specified. Note that the second parameter (aexp) may be either a player number or playfield number (see the section on PMG conventions, above). Valid **BUMPs:**

```
Player to Player:      BUMP(0-3,0-3)
Player to Playfield:   BUMP(0-3,8-11)
Missile to Player:     BUMP(4-7,0-3)
Missile to Playfield:  BUMP(4-7,8-11)
```

Note: **BUMP(p,p)**, where the p's are 0 through 3 and identical, always returns 0 (i.e. a player can't collide with itself).

Note: we advise that you reset the collision registers if you have not checked them in a long time or after you are through checking them at any given point in a program. You can do this using **HITCLR.**

HITCLR

Format: HITCLR
Example: 100 HITCLR

HITCLR resets the collision registers used by **BUMP**, thus avoiding spurious collision readings. We suggest that you use **HITCLR** just before you do something that might create a collision (move or create a P/M, change the playfield, etc.). Alternatively, you could use **HITCLR** immediately after you check for collisions (using **BUMP**).

f PMADR

Format: PMADR(pmnum)
Example: P0=PMADR(0)

The **PMADR** function returns the memory address of any player or missile. It is useful when you wish to **MOVE, POKE, BGET,** etc., data to (or from) a player area. **Note: PMADR(m)** - where m is a missile number (4 through 7) - returns the same address for all missiles.

Using POKE and PEEK with P/M's

One of the most common ways to put player data into a player stripe may well be to use **POKE**. In conjunction with **PMADR**, it is easy to write understandable player loading routines, for example:

```
10 For Loc=48 To 52
20   Read A:Poke Pmadr(0)+Loc,A
30 Next Loc
40 Data $99,$BB,$FF,$BB,$99
```

PEEK might be used to find out what data is in a particular player location.

Using MOVE with P/M's

MOVE is an efficient way to load a large player and/or move a player vertically by a large amount. This ability to **MOVE** data either upwards or downwards allows for interesting possibilities. Also, it would be easy to have several player shapes contained in stripes and then **MOVE**d into place at will. For example,

```
Move Adr(A$),Pmadr(2),128
```

could move an entire double line resolution player from A$ to player 2, and

```
Poke Pmadr(1),$ff:Move Pmadr(1),Pmadr(1)+1,127
```

would fill player 1's stripe with all "on" bits, creating a solid stripe on the screen.

Using BGET and BPUT with P/M's

As with **MOVE, BGET** may be used to fill a player memory quickly with a player shape. The difference is that **BGET** may obtain a player directly from the disk! For example,

```
Bget #3,Pmadr(0),$80
```

would get a PMG.2 mode player from the disk file **OPEN**ed on channel 3, and

```
Bget #4,Pmadr(4),$500
```

would fill all the missiles and players in PMG.1 mode - with a single statement!

BPUT would probably be most commonly used during program development to save a player shape (or shapes) to a file for later retrieval by **BGET.**

Using USR with P/M's

Because of USR's ability to pass parameters to an assembly language routine, PMG functions (written in assembly language) can be incorporated easily into to BASIC XE. For example,

```
A=Usr(Pmblink,Pmadr(2),$80)
```

might call an assembly language program (at address PMBLINK) to blink player 2, whose size is 128 bytes.

Two P/M Graphics Programs

```
100 Setcolor 2,0,0:Rem "Note: still in GR.0"
110 Pmgraphics 2:Rem "double line res"
120 Let Width=0:Y=48:Rem "initializing"
130 Pmclr 0:Pmclr 4:Rem "clear player 0 and missile 0"
140 Pmcolor 0,13,8:Rem "a nice green player"
150 P=Pmadr(0):Rem "gets address of player 0"
160 For I=P+Y To P+Y+4:Rem "a 5 element player"
170   Read V1:Rem "see below for DATA scheme"
180   Poke I,V1:Rem "actually setting up"
190 Next I
200 For X=1 To 120:Rem "player movement loop"
210   Pmmove 0,X:Rem "moves player horizontally"
220   Sound 0,X+X,0,15:Rem "just making some noise"
230 Next X
240 Missile 0,Y,1:Rem "a one-high missile at top of player"
250 Missile 0,Y+2,1:Rem "another, in middle of player"
260 Missile 0,Y+4,1:Rem "and at bottom of player"
270 For X=127 To 255:Rem "missile movement loop"
280   Pmmove 4,X:Rem "moves missile 0"
290   Sound 0,255-X,10,15
300   If (X&7)=7:Rem "every eighth horiz. position"
310     Missile 0,Y,5:Rem "you have to see this to believe it"
320   Endif :Rem "you could have had an ELSE, of course"
330 Next X
340 Pmmove 0,0:Rem "so width doesn't change on screen"
350 Width=Width+2:Rem "we'll make the player wider"
360 If Width>4 Then Width=0
370 Pmwidth 0,Width:Rem "the new width"
380 Pmclr 4:Rem "no more missile"
390 Goto 200:Rem "do it all again"
400 Rem
410 Rem "**** the player's shape DATA ****"
420 Rem "    84218421 "
430 Rem "$99 [1..11..1] "
440 Rem "$BD [1.1111.1] "
450 Rem "$FF [11111111] "
460 Rem "$BD [1.1111.1] "
470 Rem "$99 [1..11..1] "
480 Data $99,$BD,$FF,$BD,$99
```

Notice how the data for the player shape is built up - draw a picture on an 8-wide by n-high piece of grid paper, filling in whole cells. Call filled in cells '1', and empty cells '0'. Convert the 1's and 0's to hex notation and, viola! -- you have your player.

This program will run noticably faster if you use multiple statements per line. It was written as above for clarity, only.

A more complicated program, sparsely commented.

```
100 Graphics 0:Rem "not necessary, just prettier"
110 Pmgraphics 2:Pmclr 0:Pmclr 1
120 Setcolor 2,0,0:Pmcolor 0,12,8:Pmcolor 1,12,8
130 P0=Pmadr(0):P1=Pmadr(1):Rem "addr's of 2 players"
140 V0=60:Vold=V0:Rem "starting vertical pos'n"
150 H0=110:Rem "starting horizontal pos'n"
160 For Loc=V0-8 To V0+7:Rem "a 16-high double player"
170   Read X
180   Poke P0+Loc,Int(X/$0100)
190   Poke P1+Loc,X&$ff
200 Next Loc
210 Rem "animate it"
220 Let Radius=40:Deg
230 While 1:Rem "infinite loop!!"
240   C=Random(15):Pmcolor 0,C,8:Pmcolor 1,C,8
250   For Angle=0 To 355 Step 5:Rem "in DEGrees, remember"
260     Vnew=V0+Radius*Sin(Angle)
270     Vchange=Vnew-Vold:Rem "change in vpos"
280     Hnew=H0+Radius*Cos(Angle)
290     Pmmove 0,Hnew;Vchange:Pmmove 1,Hnew+8;Vchange
300     Rem "move two players together"
310     Vold=Vnew
320     Sound 0,Hnew,10,12:Sound 1,Vnew,10,12
330   Next Angle
340   Rem "just did a full circle!"
350 Endwhile
360 Rem "we better NEVER get here!"
370 Rem "**** the fancy player DATA ****"
380 Rem "       84218421|84218421 "
390 Rem "$03C0 ......##|##...... "
400 Rem "$0C30 ....##..|..##.... "
410 Rem "$1008 ...#....|....#... "
420 Rem "$2004 ..#.....|.....#.. "
430 Rem "$4002 .#......|......#. "
440 Rem "$4E72 .#..###.|.###..#. "
450 Rem "$8A51 #...#.#.|.#.#...# "
460 Rem "$8E71 #...###.|.###...# "
470 Rem "$8001 #.......|.......# "
480 Rem "$9009 #..#....|....#..# "
490 Rem "$4812 .#..#...|...#..#. "
500 Rem "$47E2 .#...###|###...#. "
510 Rem "$2004 ..#.....|.....#.. "
520 Rem "$1008 ...#....|....#... "
530 Rem "$0C30 ....##..|..##.... "
540 Rem "$03C0 ......##|##...... "
550 Rem
560 Data $03C0,$0C30,$1008,$2004,$4002,$4E72,$8A51,$8E71
570 Data $8001,$9009,$4812,$47E2,$2004,$1008,$0C30,$03C0
```

The factor slowing this program the most is the SIN and COS being calculated in the movement loop. If these values were precalculated and placed in an array this program would move!

Space For Your Notes

Sound

This chapter is devoted to the SOUND statement, and shows how to access the many forms of sound available on Atari Home Computers.

SOUND (SO.)

Format: SOUND aexp1,aexp2,aexp3,aexp4

The SOUND statement causes the specified note to begin playing as soon as the statement is executed. The note will continue playing until the program encounters another SOUND with the same aexp1 or an END. aexp1 is the voice on which you want the sound produced, and ranges between 0 and 3, inclusive. aexp2 is the frequency (pitch) of the sound, and ranges between 0 and 255, inclusive. The lower aexp2 is, the higher the frequency. aexp3 is a measure of the sound's distortion (fuzziness). Valid numbers are 0 -14, even numbers only. A value of 10 creates pure tones like a flute, and a 12 produces sounds similar to a guitar. aexp4 is the volume of the sound. Valid values are 1 - 15; the lower the number, the lower the volume.

Here is a table for various musical notes using a distortion of 10:

Note:	Low Notes		High Notes		
C	14	29	60	121*	243
B	15	31	64	128	255
B^b / $A^\#$	16	33	68	136	
A	17	35	72	144	
A^b / $G^\#$	18	37	76	153	
G	19	40	81	162	
G^b / $F^\#$	21	42	85	173	
F	22	45	91	182	
E	23	47	96	193	
E^b / $D^\#$	24	50	102	204	
D	26	53	108	217	
D^b / $C^\#$	27	57	114	230	

Middle C is marked by a "*". This program plays a C scale using the above values:

```
10 Read A:If A>255 Then End
20 Sound 0,A,10,10:Print A
30 For Wait=1 To 400:Next Wait
40 Goto 10
50 Data 14,15,16,17,18,19,21,22,23,24,26,27,29,31,33
60 Data 35,37,40,42,45,47,50,53,57,60,64,68,72,76,81
70 Data 85,91,96,102,108,114,121,128,136,144,153,162
80 Data 173,182,193,204,217,230,243,255,256
```

Notice that the DATA statement in line 80 ends with a 256, which is outside of the designated range. The 256 is used as an end-of-data marker.

Space For Your Notes

Introducing the Array Sorting Statements

Rather than go directly into the descriptions of SORTUP and SORTDOWN, we thought it best to begin with some comments and hints about their use, because they have many foibles in common.

First and foremost, note that SORTUP and SORTDOWN can only be used to sort arrays. In their simplest form they are extremely easy to use. For example, consider the following short program:

```
10 Dim Array$(5,20)
20 For I=1 To 5:Input "String) ",Array$(I;):Next I
30 Sortup Array$
40 For I=1 To 5:Print Array$(I;):Next I
50 Run
```

This program simply sorts 5 INPUTted strings and then shows the sorted order. At this time, we would like to suggest that you type in this program and try it out (Keep it around - we will use it more later). Give several different sets of words as answers. Note how neatly it sorts the words into ascending order.

Or does it? Try entering some words in uppercase and some in lowercase. What happens? Does it surprise you to find that "ZOO" comes before "apple"? Actually, the reason for this behavior is readily understood once you realize that SORTUP works on characters using ATASCII ordering (see Appendix A for a list of ATASCII codes).

Even if we restrict ourselves to the "printable" characters in the ATASCII set (alphanumeric and standard symbols), we find no real help. Digits come before uppercase letters which come before lowercase letters, but symbols are intermixed in no real useful fashion. Because the effects of this hodgepodge ordering may not be desirable in a sorted list, you may wish to limit a sort to a substring of the string elements in a savar. For example, if you have a savar where each string within it contains both a person's name and their phone number, you may wish to perform a sort based solely on names. Further, to ensure that the sorted order is consistent, you may wish to ensure that the names are stored in uppercase only.

Fortunately, SORTUP and SORTDOWN offer you the ability to sort based on substrings. And, while BASIC XE does not provide a built-in method of obtaining uppercase, non-inverse strings, it isn't very hard to build a subroutine that will do the real work for you. For example, the following PROCEDURE converts all characters in its svar parameter String$ (not a savar) to non-inverse, and converts lowercase letters to uppercase:

```
800 Procedure "To Upper" Using !String$
810   Local I,Temp
820   For I=1 To Len(String$)
830     Temp=Asc(String$(I))&$7f
840     If Temp)$60 And Temp($7b Then Temp=Temp&$5f
850     String$(I,I)=Chr$(Temp)
860   Next I
870 Exit
```

For now, don't enter this subroutine. Instead, let's investigate the concept of substrings, as mentioned above. Just change line 30 in that little program we typed in earlier so that a LIST gives you the following:

```
10 Dim Array$(5,20)
20 For I=1 To 5:Input "String) ",Array$(I;):Next I
30 Sortup Array$ Using ;3,5
40 For I=1 To 5:Print Array$(I;):Next I
50 Run
```

Once again, enter some strings in response to INPUT's prompt. This time, though, pay special attention to the third through fifth characters of each string. Notice anything funny about the sorted order? That's right, it is based solely on the characters in those positions. If you have worked with BASIC XE string arrays at all yet, the notation in line 30 may be both familiar and confusing. Perhaps changing line 40 to the following will clarify the meaning of line 30:

```
40 For I=1 To 5:Print Array$(I;3,5),Array$(I;):Next I
```

This little example should serve to remind you that you may reference characters within an element of a string array just as easily as you may reference them in an ordinary string. The "magic" character is the semi-colon. It separates the array element number from the desired character positions. (And, as the second usage of Array$ in that same line shows, the semi-colon is always necessary when referring to an element of a string array.)

Now, since the SORTUP of line 30 refers to the entire savar Array$, there is no need for the following parentheses (and, indeed, they are not allowed). Instead, the keyword USING tells BASIC XE that we will be working with only part of the array and/or its elements. In particular, the semi-colon following USING serves as a reminder that the aexps following it should be used to define a substring of the string elements in a savar.

There is one last capability of the sorting statements which we will discuss before moving on to other helpful hints. The program we have been working with seems all fine and good if we want to enter exactly five elements into the array. Suppose, though, that we did not know how many elements we'd be working with. Fear not, BASIC XE shall provide. Time for another example:

```
10 Dim String$(20,20)
20 For I=1 To 20:Input "String) ",String$(I;)
25   If Len(String$(I;)) Then Next I
30 Sortup String$ Using 1 To I-1
40 For J=1 To I-1:Print String$(J;):Next J
50 Run
```

The first change you will notice is that the FOR loop on line 20 now INPUTs 20 strings. The second change is the insertion of line 25. Instead of blindly continuing to ask for input until 20 items have been entered, the program only goes back for another if the length of the current string is non-zero. That means that you may stop entering items at any time by hitting the RETURN key alone in response to any INPUT prompt.

And look at the SORTUP in line 30. Can you guess what the Using 1 To I-1 is for? That's right, only the first I-1 elements of the array will be sorted! And if, for some reason, you wanted to never sort the first element of the array, you could have written

```
30 Sortup String$ Using 2 To I-1
```

(Why would you ever do that? Well, maybe you keep special information about a savar in its first element, thus having the actual data start at the second element.)

Well, so much for sorting string arrays. We haven't yet mentioned how to sort arithmetic arrays, but it's just as easy. You use the same statements, SORTUP and SORTDOWN, but you use the name of an arithmetic array as the first argument, like this:

```
Sortup A()
```

Notice that instead of following the array name by a dollar sign (as with string arrays), you follow it by a pair of parentheses (to indicate that the array is arithmetic). Since no element range was specified in our example, this statement will sort all elements of the array A().

If you don't want to sort the whole array, you can specify a range of elements to sort, just like we did when sorting string arrays. The following will sort elements 3 through 5, inclusive, of the array Temp() in descending order:

```
Sortdown Temp() Using 3 To 5
```

There are two restrictions to bear in mind when sorting arithmetic arrays. First, you can't specify substring indices (because numbers don't have substrings). Second, and more important, you can only sort arithmetic arrays, not matrices! Thus, if you have the following DIMension line in your program:

```
10 Dim A(40),B(10,20),C(50)
```

you could use SORTUP and SORTDOWN to sort A() and C(), but not B(), since it has two dimensions and so is a matrix.

Finally, there are a couple of rules to keep in mind:

1) The ending element number to be sorted must be greater than or equal to the beginning element number (i.e, you can't sort elements 3 TO 1),
2) Both element numbers must be within the DIMensioned bounds of the array, and
3) the previous two rules also apply to the numbers you use to specify a substring range when sorting savars.

SORTUP / SORTDOWN

Format: | SORTUP | array [USING [aexp1 TO aexp2][;aexp3,aexp4]]
| SORTDOWN |

Examples: SORTUP Aarray
SORTDOWN Aarray USING Min TO Max
SORTUP Sarray$ USING ;1,4
SORTDOWN Sarray$ USING 5 TO 10

Note: the ;aexp3,aexp4 option may be used only when sorting savars. You can not use it when sorting arithmetic arrays!

SORTUP sorts the elements of an array in ascending ATASCII or numeric order (dependent upon the array's type), while **SORTDOWN** sorts in descending order. If no element range aexp1 **TO** aexp2 is specified (1st and 3rd examples), all elements are sorted.

If an element range is specified, both beginning and ending elements must be given, separated by the keyword **TO**.

Note: if no substring ;aexp3,aexp4 is specified (4th example), the sorting is done using the string elements in their entirety. If a substring is specified, both the beginning and ending of the substring must be specified, separated by a comma. If an element range is not being used but a substring is, the keyword **USING** must precede the substring-marking semicolon (3rd example).

Note: if a string element is shorter than the specified ending position of the substring being used, the substring for that element will be shortened accordingly. If two compared strings are equal, but one is longer than the other, the longer one is greater than the shorter one (e.g., "abc"<"abcd"). This is intuitively correct as well as being consistent with the other string comparisons available in BASIC XE.

Using Fixed Data in Your Program

The three statements in this chapter allow you to insert and utilize fixed data in your BASIC XE programs. These statements are **DATA**, **READ**, and **RESTORE**.

DATA (D.)

```
Format:    DATA adata [,adata]
Examples:  100 DATA 12,13,14,15,16
           110 DATA Mike,Becky,Tommy,Kathleen
           120 DATA "adata with a , in it"
```

DATA is used in conjunction with **READ** to access elements in a data list. A **DATA** statement may be anywhere in a program, but it must contain at least as many adata items as used in the **READ** statement that accesses them; otherwise an "No DATA to READ" error (#6) is displayed on the screen. When more than one **DATA** statement is used, the adata items form a single list. For example, the first two examples could just as well be combined into

```
100 DATA 12,13,14,15,16,Mike,Becky,Tommy,Kathleen
```

Note: all characters except comma (,) and <RETURN> are legal in adata. However, if you put adata in double quotes ("adata"), then all characters except double quote (") and <RETURN> are allowed (as in the last example).

READ

```
Format:    READ var1 [,var2...]
Examples:  200 READ A,B,C,D,E
           210 READ A$,B$,C$,D$,E$
```

The **READ** statement is used to retrieve adata items in a **DATA** list, and store them in program variables for use. When a **READ** is executed, the first available adata item is stored in var1, the second is stored in var2, and so on. The adata item and the variable into which it is to be stored must be of the same data type (arithmetic or string).

The following program sums a group of numbers using **READ** and **DATA**:

```
10 For N=1 To 5
20   Read D:M=M+D
30 Next N
40 Print "Sum is ";M
50 End
60 Data 30,15,106,87,47
```

RESTORE (RES.)

Format: RESTORE [lineno]
Examples: 100 RESTORE
RESTORE X+2

BASIC XE uses an internal 'pointer' to keep track of the next adata item in the DATA list to be READ. When used without the optional lineno, RESTORE resets this pointer to the first adata item in the first DATA statement in the program. When lineno is specified, RESTORE sets the pointer to the first adata item in the DATA statement on the program line lineno. This permits repetitive use of the same adata items, as shown in the following example:

```
10 For N=2 To 1 Step -1
20   Restore 80+N
30   Read A,B:M=A+B
40   Print "Total is ";M
50 Next N
60 End
81 Data 30,15
82 Data 10,20
```

Accessing Memory Directly

The commands in this chapter allow you to access memory directly, and are very useful when you want to inspect and/or modify Atari variables and routines. Each of the commands in this chapter allows you to specify an optional bank number. For a discussion of the meaning of this number, see **EXTEND**.

The statements discussed here are **POKE**, **DPOKE**, and **MOVE**, and the functions are **PEEK** and **DPEEK**.

f PEEK

```
Format:    PEEK(aexp [,bank])
Examples:  1000 IF PEEK($4000,4)=255 THEN PRINT "Main Memory $4000=255"
           100 PRINT "Left Margin is "; PEEK(82)
```

PEEK Returns the value stored at memory location **aexp**. The address specified must evaluate to an integer between 0 and 65535. The value returned will be a decimal integer between 0 and 255, inclusive. This function allows you to examine either **RAM** or **ROM** locations. In the first example above, **PEEK** is used to determine whether location $4000 in main memory contains the value 255. In the second example, **PEEK** is used to find the current left margin.

POKE

```
Format:    POKE aexp1,aexp2 [,bank]
Examples:  POKE 82,10
           100 POKE 82,20
```

The **POKE** statement puts the value **aexp2** into memory location **aexp1**. **aexp1** may range in value between 0 and 65535, inclusive, and **aexp2** has range 0..255. The first example changes the screen's left margin from its default value of 2 to a new value of 10. To restore the margin to its normal default position, press <SYSTEM RESET>.

Note: POKE cannot be used to alter **ROM** locations.

While you are becoming familiar with this statement we advise that you first **PEEK** at the memory location and write down the value before you **POKE** in a new value. Then, if the **POKE** doesn't work as anticipated, you can **POKE** the original value back in.

f DPEEK

Format: DPEEK(aexp [,bank])
Example: PRINT "Variable Name Table is at ";DPEEK($82)

DPEEK is very similar to the **PEEK** function, except that it allows you to find out the two-byte value at the memory locations aexp and aexp+1. This is especially useful when looking at locations which contain address information, as in the above example. If you did this example using **PEEK**s, it would look like

```
Print "Variable Name Table is at ";Peek(130)+Peek(131)*128
```

It's obvious that using **DPEEK** is much easier.

DPOKE

Format: DPOKE aexp1,aexp2 [,bank]
Example: DPOKE 88,$8000

DPOKE is similar to **POKE**, except that it allows you to put a two-byte value into memory locations **aexp1** and **aexp1+1**. **aexp2** is the value, and must be an integer value 0..65535, inclusive. In the above example, the address of the upper left-hand corner of the screen (this address is stored at locations 88 and 89) is changed to $8000. To do this using **POKE**s you would need to do an amazing amount of math to get the right number into each of the two bytes.

MOVE

Format: MOVE aexp1,aexp2,aexp3 [,bank]
Example: MOVE $D000,$8000,$400

Caution: be careful with this command! **MOVE** will move any number of bytes from any address to any address at assembly language speed. No address checks are made! **aexp1** is the starting address of the block you want to move, **aexp2** is the starting address of the place where you want the block moved to, and **aexp3** is the length of the block. The sign of **aexp3** (the length) determines the order in which the bytes are moved, as follows:

Positive		Negative	
(from)	-> (to)	(from+len-1)	-> (to+len-1)
(from+1)	-> (to+1)	(from+len-2)	-> (to+len-2)
:	:	:	:
(from+len-1)	-> (to+len-1)	(from)	-> (to)

When the length is positive, the destination block can overwrite lower part of the source block. When the length is negative, the destination block can overwrite the upper part of the source block.

Note: MOVE cannot automatically move memory between banks. To do so you must first **MOVE** the block to main memory and then **MOVE** it to the other bank.

Arithmetic Functions

The arithmetic functions supported by BASIC XE are **ABS, INT, SGN, SQR, EXP, LOG, CLOG, RND,** and **RANDOM.** At the end of the chapter you will find a program that shows these functions in use.

f ABS

Format: ABS(aexp)
Example: A=ABS(-160)

ABS returns the absolute (positive) value of aexp.

f INT

Format: INT(aexp)
Examples: I=INT(-3.445)
X=INT(14.753)

INT returns the greatest integer less than or equal to aexp. This is true whether the expression evaluates to a positive or negative number. Thus, in the first example, -4 is assigned to I, and 14 is assigned to X in the second example. Note: this function should not be confused with the INT function on calculators which simply truncates all decimal places. For those of you with a mathematical background, you may think of INT as the "Floor" function.

f SGN

Format: SGN(aexp)
Example: 100 X=SGN(-100)

SGN returns a -1 if aexp evaluates to a negative number, a 0 if aexp evaluates to 0, or a 1 is aexp evaluates to a positive number.

f SQR

Format: SQR(aexp)
Example: X=SQR(100)

SQR returns the square root of aexp. Note: aexp must be positive.

f EXP

Format: EXP(aexp)
Example: PRINT EXP(3)

The **EXP** function returns the value of **e** (approximately 2.71828179), raised to the power aexp (i.e., e^{aexp}).

f LOG

Format: LOG(aexp)
Example: A=LOG(20)

The **LOG** function returns the natural logarithm (ln) of aexp. **LOG(0)** gives an error, and **LOG(1)** is 0.

Note: **LOG** and **EXP** are complementary functions (i.e., both **LOG(EXP(n))** and **EXP(LOG(n))** equal n, within the bounds of the accuracy of BASIC XE's math routines).

f CLOG

Format: CLOG(aexp)
Example: A=CLOG(10)

The **CLOG** function returns the base 10 logarithm ($\log_{10}$) of aexp. **CLOG(0)** gives an error, and **CLOG(1)** is 0.

f RND

Format: RND(aexp)
Example: 10 X=RND(0)

RND returns a hardware-generated random number greater than or equal to 0, but less than 1. aexp is a dummy and has no effect on the number returned, but is required anyway.

f RANDOM

Format: RANDOM(aexp1[,aexp2])
Examples: X=RANDOM(99)
Y=RANDOM(10,20)

The **RANDOM** function returns a random integer dependent upon aexp1 and aexp2. When aexp1 alone is specified (as in the first example), the value returned is between 0 and aexp1-1, inclusive. When both aexp1 and aexp2 are specified (as in the second example), the value returned is between aexp1 and aexp2, inclusive.

An Example Program Using Arithmetic Functions

```
500 Console=$d01f:Start=$01
510 Open #1,4,0,"K:"
520 Test=-2.71828183
530 Print :Print "We start with a value of ";Test
540 Test=Abs(Test)
550 Print :Print "Its absolute value is ";Test
560 Test=Int(Test)
570 Print :Print "And the integer part of that is ";Test
580 Test=Sqr(Test)
590 Print :Print "Which has a square root of ";Test
600 Test=Test/2
610 Print :Print "Half of that gives ";Test
620 Print " [remember that number, half SQR(2)]"
630 Test=Sgn(Test)
640 Print :Print "The 'SGN' of that is ";Test
650 Deg
660 Test=Atn(Test)
670 Print :Print "Whose ArcTangent of ";Test;" is"
680 Test=Int(Test)
690 Print " close. Correct result is ";Test;" degrees"
700 Print :Print "The sine and cosine of ";Test;" degrees:"
710 Print "      sine = ";Sin(Test)
720 Print "    cosine = ";Cos(Test)
730 Print " [look at the number you remembered]"
740 Print :Print "hit  START  for next part←";
750 While Peek(Console)&Start:Endwhile
760 Graphics 0
770 Test=Clog(100)
780 Print "The common (base 10) log of 100 is ";Test
790 Test=Log(Test)
800 Print :Print "Which has natural log of ";Test
810 Test=Exp(Test)
820 Print :Print "'e' is the base of the natural logs,"
830 Print "  and e to that power is ";Test
840 Print :Print " [which is pretty darn close to 2]"
850 Print :Print "Hit any key to continue...";
860 Get #1,Key
870 Graphics 0
880 Print :Print "Now lets flip some coins, using that"
890 Print "  value as 1 greater than the maximum"
900 Print "  pseudo-random value we want!":Print
910 Count=0
920 While Abs(Count)<3
930   If Random(Test):Count=Count+1:Print ," Heads"
940     For V=12 To 0 Step -0.2:Sound 0,10,2,V:Next V
950   Else :Count=Count-1:Print ,"  Tails  "
960     For V=15 To 0 Step -0.25:Sound 0,80,12,V:Next V
970   Endif
980 Endwhile
990 If Count>0:Print "        [ Heads won ]"
1000 Else :Print "       [ Tails won ]"
1010 Endif
```

Space For Your Notes

Trigonometic Functions

Discussed in this chapter are the trigonometic functions **COS**, **SIN**, and **ATN**, and the statements **DEG** and **RAD**. Also included is a table that shows you how to get other trascendental trig functions using the ones provided.

DEG / RAD

Format: DEG
RAD

These two statements allow you to specify whether the angles used in the trig functions are in DEGrees or RADians. Note: BASIC XE defaults to radians. Also, all trig functions following a **DEG** or **RAD** are performed using that angle measurement until the mode is changed by another **RAD** or **DEG**, respectively.

f COS

Format: COS(aexp)
Example: 100 PRINT COS(0)

COS returns the cosine of aexp. The operation is done in radians or degrees, dependent upon whether **DEG** or **RAD** has been most recently used.

f SIN

Format: SIN(aexp)
Example: 100 X=SIN(0)

The **SIN** function returns the sine of aexp. The operation is done in degrees or radians, dependent upon whether **DEG** or **RAD** has been most recently used.

f ATN

Format: ATN(aexp)
Example: 100 X=ATN(1)

ATN returns the arctangent (Tan^{-1}) of aexp. The operation is done in degrees or radians, dependent upon whether **DEG** or **RAD** has been most recently used.

A Table of Derived Functions

The following table lists some of the trigonometric and hyperbolic functions you can derive from the arithmetic and trigonometric functions available in BASIC XE. The term "x" is the value on which you wish to perform the derived function, and is simply an aexp. Also, you will see "C" in some of the functions. This is a constant dependent upon whether the angles are measured in degrees or radians. C=90 in DEGree mode, and C=1.57079633 (pi/2) in RADian mode.

Trigonometric Function	Derivation
Tangent	SIN(x)/COS(x)
Cotangent	COS(x)/SIN(x)
Secant	1/COS(x)
Cosecant	1/SIN(x)
ArcSine (Sin^{-1})	ATN(x/SQR(1-x^2))
ArcCosine (Cos^{-1})	-ATN(x/SQR(1-x^2))+C
ArcCotangent (Cot^{-1})	ATN(x)+C
ArcSecant (Sec^{-1})	ATN(SQR(x^2-1))+(SGN(x-1)*C)
ArcCosecant (Csc^{-1})	ATN(1/SQR(x^2-1))+(SGN(x-1)*C)

Hyperbolic Function	Derivation
SineH	(EXP(x)-EXP(-x))/2
CosineH	(EXP(x)+EXP(-x))/2
TangentH	-EXP(-x)/(EXP(x)+EXP(-x))*2+1
CotangentH	EXP(-x)/(EXP(x)-EXP(-x))*2+1
SecantH	2/(EXP(x)+EXP(-x))
CosecantH	2/(EXP(x)-EXP(-x))
ArcSineH ($SinH^{-1}$)	LOG(x+SQR(x^2+1))
ArcCosineH ($CosH^{-1}$)	LOG(x+SQR(x^2-1))
ArcTangentH ($TanH^{-1}$)	LOG((1+x)/(1-x))/2
ArcCotangentH ($CotH^{-1}$)	LOG((x+1)/(x-1))/2
ArcSecantH ($SecH^{-1}$)	LOG((SQR(1-x^2)+1)/x)
ArcCosecantH ($CscH^{-1}$)	LOG((SGN(x)*SQR(x^2+1)+1)/x)

BASIC XE and Machine Language Subroutines

A subroutine is simply a piece of a program that accomplishes a single task. This means that a program is really just a bunch of subroutines strung together. But what if you want to execute the same subroutine a bunch of times? You could type it in every time you want to use it, but that could mean a lot of boring typing. The solution is to use one of BASIC XE's special subroutine calls. They all allow you to write a subroutine once, and then have it get executed several times in different parts of your program.

How you get a subroutine executed (i.e., how you call a subroutine) depends upon the type of subroutine you are using. The GOSUB subroutine structure lets you call a BASIC subroutine by line number, the USR function lets you call a machine language subroutine by address, and PROCEDURE allows you to call a BASIC subroutine by name! Since each of these subroutine structures is different, they are discussed in depth in separate sections, starting with the easiest to understand, GOSUB.

GOSUB (GOS.)

Format: GOSUB lineno

GOSUB allows you to 'call' an unnamed subroutine written in BASIC XE. lineno specifies the starting line number of the subroutine. A GOSUB subroutine must end with a RETURN or EXIT (if you use LOCAL avars within the subroutine) so that program execution may continue with the statement after the GOSUB.

To prevent accidental triggering of a subroutine whose code follows the main program, place an END statement between the end of the program and the start of the subroutine.

Caution: Like the FOR and WHILE statements, GOSUB uses the program stack to save its return lineno. If the subroutine is not allowed to complete normally (e.g., you exit via a GOTO) the return lineno must be POPped off the stack or it will cause an error. Also, if you use LOCAL avars within a GOSUB subroutine and do not exit via EXIT, you must POP the previous avar values off the stack yourself.

RETURN (RET.)

Format: lineno RETURN

RETURN is used to exit a GOSUB subroutine that does not contain LOCAL avars. If the subroutine does use LOCAL, you must end it with an EXIT.

When you RETURN from a GOSUB, program execution continues at the statement after the GOSUB call.

Introducing PROCEDURE and its Related Statements

Before describing the individual statements used to create and call named subroutines, we present an introduction to them because they are interdependent, and we felt that having a small but effective demonstration of their use would make it easier to understand the later definitions.

If you have programmed at all in any dialect of BASIC, you have used the GOSUB...RETURN construction. For example, you might see a program like the following (This program is for demonstration purposes only, but it is a fairly amusing little thing to spring on an unsuspecting friend):

```
20 Value=100
30 Min=10:Max=90:Gosub 100
40 Result1=Num
50 Min=10*Value:Max=90*Value:Gosub 100
60 Result2=Num
70 If Result2>Value*Result1 Then 90
80 Print "You appear to be conservative":End
90 Print "You seem ready to take risks":End
100 Rem "The Subroutine"
110 Print :Print "Please give me a number between"
120 Print Min;" and ";Max;
130 Input ", inclusive> ",Num
140 If Num>=Min And Num<=Max Then Return
150 Inverse :Print "Can't you read?  That number is"
160 Print "  out of the range I gave you. ":Normal
170 Goto 100
```

In a small program like this one, the GOSUB may be just fine. As programs get larger, though, lines like GOSUB 3250 become less and less meaningful. Atari BASIC (and thus BASIC XE) allows you to do something like this:

```
10 Let Getinrange=100
20 Value=100
30 Min=10:Max=90:Gosub Getinrange
```

By giving a name to the subroutine, we can make our code more readable. A disadvantage to this method is that BASIC XE (in common with Atari BASIC) allows only 128 unique variable names. Using a variable name as a subroutine name diminishes the pool of available names. This, then, is the first advantage of BASIC XE's new procedures: we use string constant to name them, so we need waste no variable names! Look at the listing opposite -

```
20 Temp=100
30 Call "Get In Range" Using 10,90 To Result1
50 Call "Get In Range" Using 10*Temp,90*Temp To Result2
70 If Result2<Temp*Result1:Type$="conservative"
80 Else :Type$="a risk taker"
90 Endif
95 Print  Using "You seem to be XXXXXXXXXXXX/.",Type$:End
100 Procedure "Get In Range" Using Min,Max
110   Local Temp:Temp=1e+90
120   While Temp<Min Or Temp>Max
130     If Temp<>1e+90:Print
140       Inverse :Print "Can't you read?  That number is"
150       Print "  out of the range I gave you. ";:Normal
160     Endif
170     Print :Print "Please give me a number between"
180     Print Min;" and ";Max;
190     Input ", inclusive) ",Temp
200   Endwhile
210 Exit Temp
```

Confused? Not too surprising. Let's take a look at the new lines a step at a time. First, in line 30, note the **CALL** to the **PROCEDURE** named "Get In Range". See how clear accessing this subroutine is, since we can use any characters we like in the name string. That's pretty easy, right?

But what about the **USING** that appears in both the **PROCEDURE** and **CALL** statements? In line 30, we are 'using' values of 10 and 90. But in line 100, we are 'using' the variables **Min** and **Max**. Isn't that neat? We didn't have to assign the values 10 and 90 to **Min** and **Max** before we called the subroutine: **CALL** does the work for us! This is called 'passing parameters' to a procedure.

It gets better. Notice the **EXIT** statement of line 210. It allows the procedure to return a value (the contents of Temp) to the **CALL**. The value is placed into the variable that follows the **TO** in the **CALL** statement (Result1, in this case). That's reasonable, right? If you can 'pass' parameter values, you should be able to 'return' parameter values. But doesn't using the variable Temp in the procedure subroutine wreak havoc on its later use in the main program (e.g., in line 50)?

Ah, but there's line 110, with its deceptively simple-looking **LOCAL Temp** statement. By using it we have created a 'private' copy of Temp for use in the procedure. Any changes to Temp between the **LOCAL** and the **EXIT** won't affect its value in the rest of the program. Wow!

The example we just worked through uses all of the new procedure-oriented statements: **PROCEDURE**, **CALL**, and **EXIT**. By no means, though, did we use all of the capabilities of these statements.

PROCEDURE (PROC.)

Format: PROCEDURE pname [USING rvar1 [,rvar2...]]
Examples: 1000 PROCEDURE "Calculate Pay" USING Hours,Rate,!Taxtable()
387 PROCEDURE "Print Msg" USING !Msg$
4040 PROCEDURE "Quit"

Note: if **rvar** is an mvar, svar, or savar, it must be preceded by an exclamation point (!). See **rvar** in the glossary for more info.

The **PROCEDURE** statement is the nucleus around which named subroutines in BASIC XE are built. It defines the beginning of a subroutine which will be terminated by **EXIT**, and executed via **CALL**.

pname is the name of the **PROCEDURE**, and is simply a valid string constant. In the examples above you can see that spaces have been used in the **pnames** to add clarity to the program. As a matter of good programming style, you use names that describe what the **PROCEDURE** does, shortening them only if you begin to run out of memory.

When you **CALL** a **PROCEDURE**, the return lineno is pushed onto the BASIC XE stack so that execution can continue with the statement following the **CALL** when the **PROCEDURE** is done.

If you pass parameters to the **PROCEDURE** (via **USING**), **CALL** will push the current 'values' of **rvar1, rvar2,...** onto the stack, then put the **pexp1, pexp2,...** 'values' (see **CALL**) into the receiving variables, and finally pass control to the **PROCEDURE**. This is a fairly straightforward process when the rvars are avars, because the 'values' pushed onto the stack are simply numeric constants. Take the following set of statements as an example:

```
10 Junk=20
20 Call "Test" Using 12*17
30 Print Junk
40 End
70 Procedure "Test" Using Junk
80   Print Junk
90 Exit
```

In this example, when the **PROCEDURE** named "Test" at line 70 is **CALL**ed, the current value of the rvar **Junk** (20, as assigned in line 10) is pushed on the stack. Then the value of the pexp (12*17, or 204) is copied into **Junk**. Any subsequent references to **Junk** within the **PROCEDURE** will find that it contains this new value. For example, the **PRINT** on line 80 will display the value 408. When the **EXIT** on line 90 is executed, it will restore **Junk** to its prior value of 20, thus the **PRINT** on line 30 will display the value 20.

All that this means is that **USING** (when used in conjunction with **CALL** and **PROCEDURE**) does an implicit **LOCAL**. The purpose of this might not be perfectly clear. Thanks to the implicit **LOCAL**, we can reuse the variable name **Junk** in our procedure and so conserve on names (remember, we are allowed only 128) without worrying about changing it within the procedure. The second advantage is more difficult to see from this simplistic example: we are able to pass values into the procedure without knowing what variable names are used within it.

The example in the previous section shows this feature to some advantage, and demonstrates how the resultant code can be both smaller and more readable.

When the rvars are not avars (i.e. they're mvars, svars, or savars), the methodology is the same, but the results are more complex. The difficulty lies in understanding just what the 'value' that gets pushed on the stack is. A journey inside BASIC XE is required to answer this question. In BASIC XE the value of any variable is the contents of its entry in the Variable Value Table. This table reserves eight (8) bytes per variable - a flag byte, the variable's number (0..127), and six bytes of 'information'.

For simple avars, the 'information' is the numeric value of the variable. For svars, savars, and mvars, the flag byte indicates that the 'information' is the address and characteristics of the actual data. For example, an svar needs information about its address, its DIM length, and its current LEN length. The string data itself is located at the given address. The 'information' for both mvars and savars consists of an address and two DIMensions.

Thus, when CALL pushes the 'value' of a rvar that's a svar, savar, or mvar on the stack, it is pushing this special information. Similarly, when CALL copies a pexp that's a svar, savar, or mvar into one of these types of rvars, it is not copying the actual string or array. Instead, it is copying the special information. This is the reason that rvar and pexp require the ! prefix when they refer to these types of variables. Consider this sequence:

```
10 Fun$="Swimming is fun.":X$="Right?"
20 Call "What Fun" Using !Fun$
30 Print Fun$,X$
40 End
50 Rem "The Procedure"
60 Procedure "What Fun" Using !X$
70   Print Fun$,X$
80   X$(1,5)="Laugh"
90 Exit
```

Hopefully, you will actually try this little program. If so, you will find that line 70 shows that, as we have described above, the 'value' of Fun$ has been copied into X$. The PRINT in line 70 will display

```
Swimming is fun.        Swimming is fun.
```

The real surprise comes when the PRINT in line 30 is executed (following the successful EXIT in line 90). The resultant display is

```
Laughing is fun.        Right?
```

Do you see why? If the 'value' of Fun$ is copied to X$, then the address of Fun$ is now in X$'s entry in the Variable Value Table. Thus, any change we make to X$ affects affects the contents of Fun$. Complicated, yes?

A similar action place takes place when a savar or mvar is passed as a parameter - changes to the rvar within the PROCEDURE will affect the pexp variable in the CALL.

Technical Note: in computer lingo, avars passed to a procedure via a 'call by value', while the other types of variables are passed via a 'call by reference'.

Notes and Warnings Regarding PROCEDURE

Note: BASIC XE insists that paired pexps and rvars be of the same type. For example, the following will cause error 24 ("USING Type Mismatch"):

```
400 Call "Oh No!" Using 33
 !
720 Procedure "Oh No!" Using !A$
```

Note: BASIC XE does not make sure that you have the same number of rvars as pexps in a CALL to a PROCEDRE. If a CALL does pass too many pexps, the extra ones are ignored. If it passes too few, a value of zero is assigned to all remaining rvars parameters. This, in turn, can cause a type mismatch, since only avars may receive a numeric value. Exception: if the CALL passes no parameters, BASIC XE does nothing at all to the parameter passing area. This is on purpose, since passing parameters takes time. Thus, even a PROCEDURE expecting only numeric parameter(s) may report a mismatch error, since it attempts to obtain those parameters from the miscellaneous data left in the parameter area. Generally, we recommend passing the correct number of parameters unless you have a specific purpose which can use the "default" feature to a real advantage.

Note: you must be careful when changing the value of a svar passed as a parameter. Recall that the length of a svar is found in its Variable Value Table entry, and that the entry is copied intact to the PROCEDURE's rvar. If you then change the length of the rvar string within the procedure, it will indeed change the rvar's length in the table. However, when you EXIT, the rvar entry is not automatically copied back to the pexp used in the CALL! This can produce some bizarre results. To demonstrate - modify line 80 of the last example program to read

```
80 X$="Laugh":Print X$
```

Not surprisingly, the new PRINT in line 80 shows us that the contents of X$ are simply "Laugh". However, look at the display resulting from line 30:

```
Laughing is fun.        Right?
```

Do you see the problem? Changing X$ in line 80 changed the contents of Fun$, but it did not change the length of Fun$. Presumably, this could be a feature under the right circumstances, but there are stranger consequences possible. For example, try changing line 80 to read

```
80 X$="XXX"
```

Now line 30's PRINT will display

```
XXXming is fun.         Right?
```

which is almost surely not we wanted.

One solution to this situation is simply to avoid changing a passed string within a procedure block. This may not be satisfactory, though, so we have provided another mechanism which you can use to circumvent the problem. Change lines 20 and 90 in the original program to read

```
20 Call "What Fun" Using !Fun$ To !Fun$
90 Exit !X$
```

Using the TO guarantees that the complete new "value" of X$ will be copied back to Fun$. On this same topic, you may be relieved to know that this difficulty with length does not exist with mvars or savars.

Warning: one way to get in real trouble with either strings or arrays is to pass one back (via EXIT) which was not passed in (via CALL). Examine the following program excerpt:

```
100 Call "Oops" To !A$
110 Call "Oops" To !B$
120 Print A$,B$:End
300 Procedure "Oops"
310   Input "Type something> ",Line$
320 Exit !Line$
```

If you type in and RUN this program, giving different responses when you are prompted, you will be surprised at the results of the PRINT of line 120: A$ and B$ will be identical (up to the length of the shorter), taking on the value of the second INPUT. If you recall our discussion of what actually gets passed when a string or array is involved, this seemingly bizarre result can be explained.

When Line$ gets passed back, what is actually transferred is its Variable Value Table entry, first to A$, and then to B$. But the table entry consists (among other things) of LINE$'s address. Thus you end up with all three variables pointing to the same piece of memory!

The proper solution is to pass a string both in via USING and back out via EXIT. For savars and mvars, you need only pass the value in, since anything the PROCEDURE does these variable types is properly reflected in the original variable.

The only way you can get in trouble with arrays is if you pass an unDIMensioned array to a procedure which then DIMensions it. Unless you pass back the "value" via EXIT (similar to the fix for strings just given above), the space DIMensioned within the procedure is lost, since no variable's entry will refer to it after the EXIT is executed.

Warning: PROCEDURE must be the first statement on a line. CALL cannot find a PROCEDURE if is not at the beginning of a line. Strange and wondrous (and woefully unpredictable) things can happen if you violate this rule. Similarly, you should never allow a program to "fall through" to a PROCEDURE. Always make sure that the program immediately preceding each PROCEDURE finishes with a GOTO, STOP, END, RETURN, or EXIT. We recommend grouping all procedures at one spot in your program, preceded by an END statement.

EXIT

```
Format:    EXIT [pexp1 [,pexp2...]]
Examples:  390 EXIT 10*Maxvalue
           799 EXIT Flag,!Names$
           24990 EXIT !Inverse(),Rows,Columns
           835 EXIT
```

Note: if pexp is an mvar, svar, or savar, it must be preceded by an exclamation point (!). See pexp in the glossary for more more info.

If you have been reading this manual front to back you have encountered several examples of the statement **EXIT** by now. If you have not, we refer you to the three previous sections for some illustrative examples.

EXIT performs the following three functions:

1) If there are any variables on the stack (i.e., if you passed parameters or used **LOCAL**) **EXIT** restores them to their proper places in the Variable Value Table.
2) If there are any pexps after the **EXIT**, it places them into the rvars following the **TO** in the **CALL** statement.
3) **EXIT** checks to see whether the current subroutine was invoked via **CALL** or **GOSUB**. If it was a **GOSUB**, **EXIT** simulates the action of a **RETURN**.

Warning: no error will result if an **EXIT** statement tries to pass pexps back to a **GOSUB**. Instead, they are simply ignored. Similarly, if you pass back too many pexps to a **CALL**, the excess ones will be ignored. This design allows a single **PROCEDURE** to serve more than one function, returning more values to some **CALL**ers than to others. Remember, though, that all rvars expected by the **TO** portion of a **CALL** statement must be matched by type by the pexps of **EXIT**.

Warning: because **POP** is smart enough to pop variable 'values' off the stack, you can leave subroutines with **LOCAL** avars and/or parameters without using **EXIT**. You must, however, make sure that you **POP** all variables off the stack, as well as **POP**ping the return lineno.

CALL

Format: CALL cname [USING pexp1[,pexp2...]] [TO rvar[,rvar...]]
Examples: 10 CALL "Test"
720 CALL "Totals" USING !Values() TO Sum
800 CALL "Get Num" TO Number
100 CALL Proc$ USING 7,!A$ TO Result

Note: if rvar or pexp is an mvar, svar, or savar, it must be preceded by an exclamation point (!). See **rvar** and **pexp** in the glossary for more more info.

The **CALL** statement has been both discussed and demonstrated earlier in this chapter. In this section, then, we will not dwell on such things as the mechanics of parameter passing. Rather we will discuss the subtleties of the **CALL** statement itself.

First, unlike a **PROCEDURE** statement, the name specified by a **CALL** may be a svar instead of being a string constant (see the last of the above example lines). However, you have no other choice of format than that shown. You may use neither a substring nor an element of a string array as a **CALL**ed name. This is not an onerous restriction, though, since the great bulk of your **CALL**s will probably be made with string constants. For those rare occasions when you wish to choose one of several **PROCEDURE**s based on the value of some index, may we suggest a program format similar to the following:

```
30 Input "Give me an Index> ",Index
40 Name$=Proc$(Index;):Call Name$
```

Note: the name that you **CALL** with (whether constant or variable) must match exactly that given in a **PROCEDURE** statement. All characters are considered in the match, with upper case, lower case, and inverse video all distinct.

Caution: we remind you of the possible problem associated with using a svar as a pexp: if its length is modified in the procedure, the change is not reflected in the svar unless **TO** is used. Similarly, any array that's not **DIM**ensioned at the time of the **CALL** should receive the same treatment.

Technical Note: the number of levels you may nest **CALL**s is limited only by the amount of **FRE**e memory left for stack use. Like **GOSUB** and **WHILE**, **CALL** uses four (4) bytes of stack space, and each parameter passed occupies 12 bytes.

Note: **CALL**s are slow in comparison to **GOSUB** lineno in **FAST** mode. However, when compared to normal **GOSUB**s in slow mode, they may actually be just a bit faster if they don't pass parameters. Parameter passing can, indeed, slow things down remarkably. But, when you compare it to the method of doing several assignments before a **GOSUB**, followed by one or more afterward, it may actually save time in some situations.

f USR

Format: USR(aexp1[,aexp2...])
Example: 100 RES=USR(ADDR,A*2)

The USR function returns the result of a machine-language subroutine. aexp1 must be an integer, and is used as the address of the machine language routine to be performed. The input arguments aexp2, aexp3,... are optional, and are used as parameters to the machine language subroutine. These aexps must be between 0 and 65535, and will be rounded to the nearest positive integer if they are fractional. They are then pushed on the hardware stack in the reverse of the order given, so the machine language program may then pull them in proper forward order. Additionally, a one byte count of parameters is pushed onto the stack last, and must be popped by the USR routine. This may be changed using the SET 8,aexp.

Also, if all arguments are properly pulled from the stack, then the USR routine may return to BASIC XE simply by executing an RTS instruction. Finally, the routine may return a single 16-bit value to BASIC XE (as the "value" of the function) by placing a result in FR0 and FR0+1 ($D4 and $D5) before returning.

Note: see ADR if your machine language subroutine is in a string, as this might be problematic if you are in EXTENDed mode.

The following example uses a USR routine to ASL a number (the argument to the USR routine) and then return that value to BASIC XE.

BASIC XE statement:

```
Xasl=Usr($680,X)
```

USR routine at $680:

```
100         PLA         ;Get # of parameters
110         CMP #1      ;If not 1 EXIT
120         BNE END
130         PLA         ;MSB
140         TAX         ;Save it
150         PLA         ;LSB
160         ASL A       ;ASL LSB
170         STA $D4     ;Save it
180         TXA         ;Get MSB
190         ROL A       ;ROL it to get carry
200         STA $D5     ;Save it
210 END     RTS
```

NORMAL Video

Dec	Hex	Chr	Keystroke	Dec	Hex	Chr	Keystroke
0	$00	♥	CTRL ,	64	$40	@	SHIFT @
1	$01	├	CTRL A	65	$41	A	A
2	$02	┘	CTRL B	66	$42	B	B
3	$03	┘	CTRL C	67	$43	C	C
4	$04	┤	CTRL D	68	$44	D	D
5	$05	┐	CTRL E	69	$45	E	E
6	$06	╱	CTRL F	70	$46	F	F
7	$07	╲	CTRL G	71	$47	G	G
8	$08	◢	CTRL H	72	$48	H	H
9	$09	▗	CTRL I	73	$49	I	I
10	$0A	▙	CTRL J	74	$4A	J	J
11	$0B	▝	CTRL K	75	$4B	K	K
12	$0C	▘	CTRL L	76	$4C	L	L
13	$0D	▔	CTRL M	77	$4D	M	M
14	$0E	▁	CTRL N	78	$4E	N	N
15	$0F	▖	CTRL O	79	$4F	O	O
16	$10	┼	CTRL P	80	$50	P	P
17	$11	▏	CTRL Q	81	$51	Q	Q
18	$12	─	CTRL R	82	$52	R	R
19	$13	┼	CTRL S	83	$53	S	S
20	$14	●	CTRL T	84	$54	T	T
21	$15	▀	CTRL U	85	$55	U	U
22	$16	▌	CTRL V	86	$56	V	V
23	$17	┴	CTRL W	87	$57	W	W
24	$18	▛	CTRL X	88	$58	X	X
25	$19	▌	CTRL Y	89	$59	Y	Y
26	$1A	▙	CTRL Z	90	$5A	Z	Z
27	$1B	↰	ESC ESC	91	$5B	[	SHIFT [
28	$1C	↑	ESC CTRL ↑	92	$5C	\	SHIFT \
29	$1D	↓	ESC CTRL ↓	93	$5D	]	SHIFT]
30	$1E	←	ESC CTRL ←	94	$5E	^	SHIFT ^
31	$1F	→	ESC CTRL →	95	$5F	_	SHIFT _
32	$20	space	SPACE BAR	96	$60	♦	CTRL .
33	$21	!	SHIFT !	97	$61	a	a
34	$22	"	SHIFT "	98	$62	b	b
35	$23	#	SHIFT #	99	$63	c	c
36	$24	$	SHIFT $	100	$64	d	d
37	$25	%	SHIFT %	101	$65	e	e
38	$26	&	SHIFT &	102	$66	f	f
39	$27	'	SHIFT '	103	$67	g	g
40	$28	(	SHIFT (	104	$68	h	h
41	$29	)	SHIFT)	105	$69	i	i
42	$2A	*	*	106	$6A	j	j
43	$2B	+	+	107	$6B	k	k
44	$2C	,	,	108	$6C	l	l
45	$2D	-	-	109	$6D	m	m
46	$2E	.	.	110	$6E	n	n
47	$2F	/	/	111	$6F	o	o
48	$30	0	0	112	$70	p	p
49	$31	1	1	113	$71	q	q
50	$32	2	2	114	$72	r	r
51	$33	3	3	115	$73	s	s
52	$34	4	4	116	$74	t	t
53	$35	5	5	117	$75	u	u
54	$36	6	6	118	$76	v	v
55	$37	7	7	119	$77	w	w
56	$38	8	8	120	$78	x	x
57	$39	9	9	121	$79	y	y
58	$3A	:	SHIFT :	122	$7A	z	z
59	$3B	;	;	123	$7B	♠	CTRL ;
60	$3C	<	<	124	$7C	|	SHIFT |
61	$3D	=	=	125	$7D	♣	ESC SHIFT CLEAR
62	$3E	>	>	126	$7E	◀	ESC BK SP
63	$3F	?	SHIFT ?	127	$7F	▶	ESC TAB

INVERSE Video

Dec	Hex	Chr	Keystroke	Dec	Hex	Chr	Keystroke
128	$80		INV CTRL ,	192	$C0		INV SHIFT @
129	$81		INV CTRL A	193	$C1		INV A
130	$82		INV CTRL B	194	$C2		INV B
131	$83		INV CTRL C	195	$C3		INV C
132	$84		INV CTRL D	196	$C4		INV D
133	$85		INV CTRL E	197	$C5		INV E
134	$86		INV CTRL F	198	$C6		INV F
135	$87		INV CTRL G	199	$C7		INV G
136	$88		INV CTRL H	200	$C8		INV H
137	$89		INV CTRL I	201	$C9		INV I
138	$8A		INV CTRL J	202	$CA		INV J
139	$8B		INV CTRL K	203	$CB		INV K
140	$8C		INV CTRL L	204	$CC		INV L
141	$8D		INV CTRL M	205	$CD		INV M
142	$8E		INV CTRL N	206	$CE		INV N
143	$8F		INV CTRL O	207	$CF		INV O
144	$90		INV CTRL P	208	$D0		INV P
145	$91		INV CTRL Q	209	$D1		INV Q
146	$92		INV CTRL R	210	$D2		INV R
147	$93		INV CTRL S	211	$D3		INV S
148	$94		INV CTRL T	212	$D4		INV T
149	$95		INV CTRL U	213	$D5		INV U
150	$96		INV CTRL V	214	$D6		INV V
151	$97		INV CTRL W	215	$D7		INV W
152	$98		INV CTRL X	216	$D8		INV X
153	$99		INV CTRL Y	217	$D9		INV Y
154	$9A		INV CTRL Z	218	$DA		INV Z
155	$9B	EOL	RETURN	219	$DB		INV SHIFT [
156	$9C		ESC SHIFT DELETE	220	$DC		INV SHIFT \
157	$9D		ESC SHIFT INSERT	221	$DD		INV SHIFT]
158	$9E		ESC CTRL TAB	222	$DE		INV SHIFT ^
159	$9F		ESC SHIFT TAB	223	$DF		INV SHIFT _
160	$A0		INV SPACE BAR	224	$E0		INV CTRL .
161	$A1		INV SHIFT !	225	$E1		INV a
162	$A2		INV SHIFT "	226	$E2		INV b
163	$A3		INV SHIFT #	227	$E3		INV c
164	$A4		INV SHIFT $	228	$E4		INV d
165	$A5		INV SHIFT %	229	$E5		INV e
166	$A6		INV SHIFT &	230	$E6		INV f
167	$A7		INV SHIFT '	231	$E7		INV g
168	$A8		INV SHIFT (	232	$E8		INV h
169	$A9		INV SHIFT)	233	$E9		INV i
170	$AA		INV *	234	$EA		INV j
171	$AB		INV +	235	$EB		INV k
172	$AC		INV ,	236	$EC		INV l
173	$AD		INV -	237	$ED		INV m
174	$AE		INV .	238	$EE		INV n
175	$AF		INV /	239	$EF		INV o
176	$B0		INV 0	240	$F0		INV p
177	$B1		INV 1	241	$F1		INV q
178	$B2		INV 2	242	$F2		INV r
179	$B3		INV 3	243	$F3		INV s
180	$B4		INV 4	244	$F4		INV t
181	$B5		INV 5	245	$F5		INV u
182	$B6		INV 6	246	$F6		INV v
183	$B7		INV 7	247	$F7		INV w
184	$B8		INV 8	248	$F8		INV x
185	$B9		INV 9	249	$F9		INV y
186	$BA		INV SHIFT :	250	$FA		INV z
187	$BB		INV ;	251	$FB		INV CTRL ;
188	$BC		INV <	252	$FC		INV SHIFT |
189	$BD		INV =	253	$FD		INV ESC CTRL 2
190	$BE		INV >	254	$FE		ESC CTRL DELETE
191	$BF		INV SHIFT ?	255	$FF		ESC CTRL INSERT

BASIC XE Memory Map

Below you will find a table containing the low memory locations used by BASIC XE. In the descriptions you will find the abbreviations 'AtB' and 'BXE'. They stand for 'Atari BASIC' and 'BASIC XE', respectively.

Most of these locations are documented only because they are used to delimit areas in the memory maps on the following pages. The only locations that might be of use to you are LOMEM, STOPLN, ERRSAV, and PTABW. These, however, are associated with BASIC XE commands as follows, so you need never use **PEEK** or **POKE**:

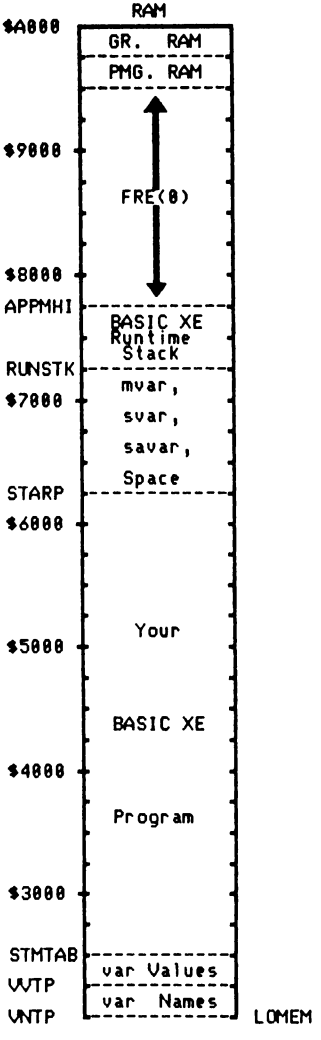
LOMEM	**LOMEM**
STOPLN	**ERR(1)**
ERRSAV	**ERR(0)**
PTABW	**SET 1,aexp**

Note: unless otherwise specified, all zero page locations $80 - $FF are used by BASIC XE.

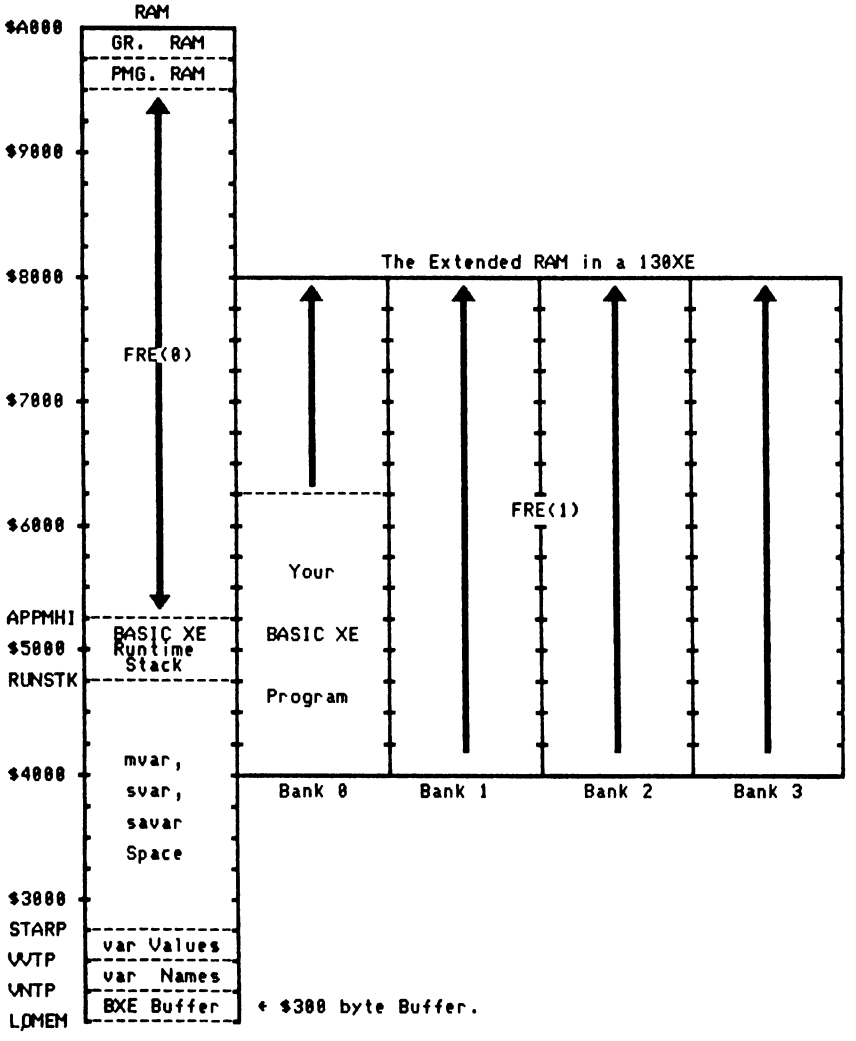
Location(s)	Label	Usage
$E-$F	APPMHI	System pointer to free memory.
$20-$2F	ZIOCB	Temporary storage for Floating Point routines.
$43-$49	FMSZPG	Temporary storage for Floating Point routines.
$80,$81	LOMEM	Low memory pointer.
$82,$83	VNTP	Variable name table pointer.
$84,$85	VNTD	Pointer to the end of variable name table plus one.
$86,$87	VVTP	Variable value table pointer.
$88,$89	STMTAB	Statement table pointer.
$8A,$8B	STMCUR	Current statement pointer.
$8C,$8D	STARP	mvar, svar, and savar value table pointer.
$8E,$8F	RUNSTK	Runtime stack pointer.
$90,$91	MEMTOP	High memory pointer.
$BA,$BB	STOPLN	Line number at which the program stopped.
$C3	ERRSAV	The number of the most recent error.
$C9	PTABW	Number of columns between tab stops.
$CB-$D1	-----	Unused by BXE!!
$D4-$D9	FR0	Floating point register 0.
$E0-$E5	FR1	Floating point register 1.
$480-$57F	-----	Used by BXE for various purposes. **Caution:** some AtB programs use this area during RUN. BXE programs that use only AtB commands can do this also, but those that take advantage of the new commands may not use this space.
$580-$67F	-----	Normally unused by BXE, but INPUT or ENTER from an external device can wipe it out.
$680-$6FF	-----	Unused by BXE!! We suggest that you use this area for your USR routines.
$700-LOMEM	-----	DOS and any other device handlers (R:, etc.) reside here. The **LOMEM** statement can change the size of this space.

Low Memory - Standard

The diagrams on this and the facing page show how BASIC XE uses memory between LOMEM and the start of cartridge memory ($A000). The diagram on this page shows how memory is used if you do not use the EXTEND statement, and the one opposite shows the memory configuration in EXTENDed mode.

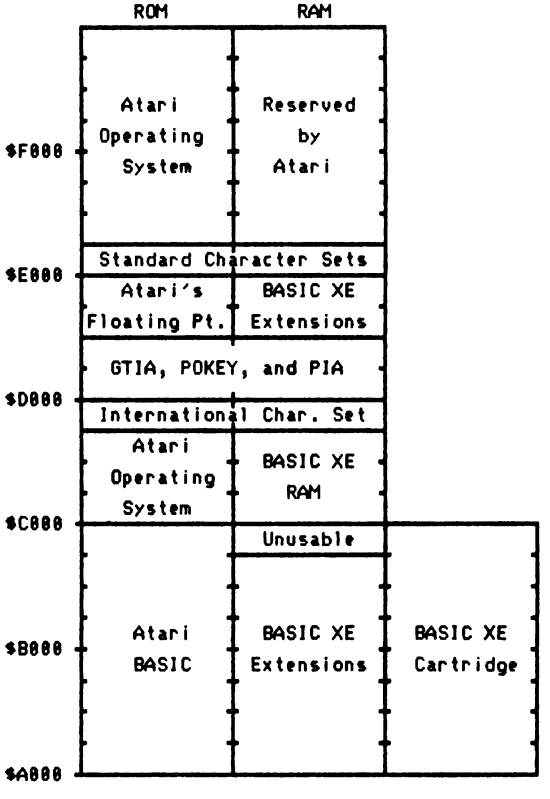


Low Memory - EXTENDed



High Memory

The diagram on this page shows the memory configuration from the start of cartridge memory to $FFFF (the end of address space). Those areas labelled 'BASIC XE Extensions' are used by BASIC XE only when you have booted using the disk extensions.



Compatability with Atari BASIC

Generally, BASIC XE is totally compatable with Atari BASIC. Virtually all programs you have written in Atari BASIC will execute properly under BASIC XE. However, there are a few subtle differences between the two BASICs, and some of these can affect whether a program will load and run or not. This appendix presents a list of known differences, but we can't guarantee that it covers all the differences.

Variable Names

When you **SAVE** or **CSAVE** a program in Atari BASIC, and then **LOAD** or **CLOAD** it into BASIC XE, you will never encounter a conflict in variable name usage. If, however, you **LIST** a program from Atari BASIC, and try to **ENTER** it into BASIC XE, you might discover that BASIC XE will not accept some lines that you know are legal in Atari BASIC.

The reason, of course, is that BASIC XE has a much larger list of commands than does Atari BASIC, and in neither BASIC can you start a variable name with a command name unless you precede it with **LET**. To illustrate how this can create a problem, consider this program line that's valid in Atari BASIC:

```
100 NUMBER=7
```

Because **NUM** is a BASIC XE statement the above line will look like

```
100 Num Ber=7
```

to BASIC XE. Since your program probably doesn't have a variable named **Ber**, the expression **Ber=7** will evaluate to zero, thus making the original statement turn into

```
100 Num 0
```

which is certainly not what you intended!

In most cases variable name conflicts will result in syntax errors, but in this particular case (and a few others) the result appears valid to BASIC XE, thus creating possibly disasterous consequences.

How can you detect and fix such problems? The easiest way is to examine a BASIC XE **LIST**ing of the program, and, thanks to BASIC XE's program formatter, the discrepancies will stick out.

Remember, however, that even **LET** will not allow you to use function names as variable names, so you need to change variable names that begin with (or match) a BASIC XE function name to something else (e.g., change BUMP to BMP or VBUMP).

Programs that RUN Too Fast

One of the reasons you bought BASIC XE in the first place was probably its speed. However, little did you realize that some of your BASIC programs (most likely games) would RUN too fast! The only solution to this is to put delays in your program. You can do this easily by CALLing a PROCEDURE that waits for some time, dependent upon the value you pass it, as follows:

```
1000 Procedure "Wait" Using Time
1010   Local Temp
1020   For Temp=1 To Time:Next Temp
1030 Exit
```

Now, just insert CALLS to this routine where you need to waste some time:

```
100 Call "Wait" Using 20
```

Memory Conflicts

BASIC XE attempts to conform to all memory location usage published in any or all of the following books:

Atari BASIC Reference Manual, by Atari, Inc.
De Re Atari, by Chris Crawford el alia
Mapping the Atari, from COMPUTE! Books
Master Memory Map, by Educational Software, Inc.

A few programs written by extemely knowledgeable individuals have made use of one or more of the following unpublished facts about Atari BASIC:

1) Atari BASIC uses certain memory locations only at certain times,
2) Certain zero-page locations have specinl meaing to Atari BASIC, and
3) Certain subroutines internal to Atari BASIC begin at certain addresses in the cartridge.

Obviously, we couldn't have added speed and features to BASIC XE without adding code and making more use of the memory reserved for BASIC. Although we kept changes to a minimum, we can't possibly be held responsible for conflicts created by programs that depend use such methods to accomplish their task. They were created specifically for use with Atari BASIC, and must remain that way.

Automatic String DIMensioning

BASIC XE will automatically DIMension strings to 40 characters for you, and this should have no effect on your Atari BASIC programs, but, if you really want to insure total compatibility, use SET 11,0.

Indented LISTings

When BASIC XE LISTs a program it automatically indents control structures (FOR, WHILE, etc.). This can be a problem if you LIST an Atari BASIC program with extemely long lines and then try to ENTER it into BASIC XE. To solve problems that arise from this, use SET 12,0.

Data Space in Extended Memory

When you use BASIC XE with an Atari 130 XE computer, there are three ways to use the "extra" 64K bytes of RAM memory which this machine gives you. Although you can use only one of these ways at a time, the flexibility is nice and may allow you to write some interesting programs. You should already be familiar with two of these ways:

1) You can use BASIC XE's **EXTEND** command to give yourself a 64K program workspace without affecting a data space of 30K bytes or more, or
2) You can boot with a DOS that allows you to use this memory as a super-fast RamDisk (Atari DOS 2.5 is a good example).

This Appendix will introduce you to the third way to use this memory.

If you don't use the memory for large programs, and if you don't use it for a RamDisk, then BASIC XE allows you to use it for your own purposes. In fact, BASIC XE has several statements and functions which were designed to help you use this memory. If you will refer to the descriptions in this manual of the following commands, you will find that each allows you to specify an optional bank number:

MOVE	POKE
BGET	DPOKE
BPUT	PEEK
	DPEEK

The bank numbers that can be used with these commands are illustrated in Appendix B. Not shown in that diagram is Bank 4, which is simply the "main" memory from $4000 to $7FFF. BASIC XE assigns it this bank number for your convenience, but in any of these commands "Bank 4" is assumed if no bank number is given.

With the exception of **MOVE**, all of these commands can be used easily and safely to store or retrieve data in any of the extended memory, so long as neither BASIC XE nor DOS is trying to use the memory at the same time. For example, you could copy a small disk file by
1) **OPEN**ing the file with its original disk inserted,
2) using **BGET** to read it into one of the banks,
3) **CLOSE**ing and re**OPEN**ing the file after inserting another disk, and
4) using **BPUT** to write the file from the extended bank. If the file is longer than 16K bytes, you could use 2, 3, or even all 4 banks to hold it while waiting for the disks to be swapped.

Use of the **MOVE** statement requires a little more care, though. The bank number you specify for a **MOVE** refers to both the source and destination addresses. Thus a command of the form

```
Move $4000,$5000,$200,3
```

would move 512 ($200) bytes from location $4000 in bank 3 to location $5000 in bank 3. This is often exactly what you want and will probably make you gloriously happy. But consider a command like this:

```
Move Adr(Goodies$),$4000,Len(Goodies$),2
```

This is dangerous and probably will not work!

If you refer to the memory map of Appendix B again, you will note that it is possible (or even probable) that BASIC XE will store your strings and arrays somewhere in the address range $4000 through $7FFF in main memory. Assume, for the moment, that the string Goodies$ is stored at address $6050. The above **MOVE** command would try to move bytes from location $6050 in bank 2 to location $4000 in bank 2. Almost certainly not what you wanted.

How can you avoid this problem? First, always **MOVE** any object that is located in main memory from $4000 to $7FFF to an intermediate location that is outside those bounds. Then **MOVE** from the intermediate location to the appropriate bank. What intermediate areas are available? If you are writing your own program from scratch, then there are several good locations available, if you will refer to Appendix B again. If you aren't using it for any other purpose, page 6 of memory ($600 to $6FF) is a good spot. Note that this limits your **MOVE**s to 256 bytes each. This may require a little work on your part, such as in this routine:

```
910 For Loc=0 To Len(X$) Step 256
920    Move Adr(X$)+Loc,$600,256
930    Move $600,$4000+Loc,256,3
940 Next Loc
```

(There is a flaw in the above program: if X$ is -- for example -- 10 characters long, then the first set of **MOVE**s will move 246 bytes too much. If this could cause a problem, your program would have to check for this situation and make a shorter **MOVE** on the last section of each string.)

The program titled "SHOWPIC" on page D-5 shows another good location to use for a **MOVE** buffer: the graphics screen memory. In this program, the screen memory is used to actually hold pictures, but there is no reason you couldn't use excess memory in this area (between APPMHI and HIMEM) for any purpose you choose.

To help get you started using extended memory in new ways, we here explain the "SHOWPIC" program, step by step. As its name implies, it shows pictures. In fact, it will show up to eight pictures in slide show fashion, and its big feature is the speed at which it shows them.

To use the program, you need two or more picture files that have been saved in what is known as "Micro-Illustrator" format. The first 7680 bytes (40 bytes per line by 192 lines) of a file in this format are simply a dump of either a GRAPHICS 24 (which is 8+16, a full screen two color mode) or GRAPHICS 31 (a full screen also, 15+16) screen memory. Most popular drawing programs for Atari 8-bit computers either use this format or provide a means of using it. For example, standard Koala Pad and Atari Artist software use a condensed format, but both allow you to produce a Micro-Illustrator file by pressing "Control-Shift-Insert" (push the Insert key while holding down both the Control and Shift keys). Doing this always produces a file of the name "PICTURE," so you must go to DOS and rename the file before you save another picture on the disk in the same way.

Since picture files in this format are large, we suggest putting the program "SHOWPIC" on a disk with nothing but DOS and the pictures. The picture files may use any 8-character name, but all must have the extension ".PIC" in order for "SHOWPIC" to find them. The paragraphs that begin on the following page explain the workings of "SHOWPIC" in some detail, and the numbers used are those of the

lines being explained.

180 The string **File$** is used only to read a line from the directory. The string array **Files$** will hold the names of up to eight files.

190 As noted above, a Micro-Illustrator picture is simply 7680 bytes "dumped" from screen memory.

200 The states of the Start, Select, and Option keys are found by **PEEK**ing location $D01F. If the Start key is pressed, the least significant bit ($01) of the location will be zero.

240 We will read a portion of the directory of the disk in drive 1. Feel free to change the drive number and/or the filename extension.

250 We will read in a maximum of 8 file names.

260,270 As we read in a filename, we check it. If there are fewer than 8 picture files on the disk, we will read the line which tells how many free sectors there are. If we find that line, we exit from the **FOR** loop early.

280,290 Because the directory listing format does not produce standard file names, we must build a proper name for later use by **OPEN**. Again, you may change the drive number and/or filename extension if you wish.

300,310 Regardless of how we exit the loop, we successfully read in one fewer than the value of the loop variable.

320 Even when you read the directory, you must close the file.

360,370 We chose a full screen black and white picture. We also chose colors which looked good on our monitor. If you are using color pictures, change to **GRAPHICS 31** and use appropriate **SETCOLOR**s.

380,390 We will read in only as many files as we found in the directory.

400 This one statement reads in the entire picture! Location $58 contains address of the beginning of screen memory (i.e., the address of the byte for **POSITION 0,0**). See any good Atari memory map book.

440 We put pictures 1 and 2 in bank 0, pictures 3 and 4 in bank 1, etc.

450,460 If it's an odd-numbered picture, we put it in the lower half of the bank. Even-numbered ones go to the top of the bank.

470 As explained above, this **MOVE** is safe because screen memory is located above $7FFF. If you use a program which somehow lowers HIMEM, this might not work!

480,490 Finish up with this file and loop for the next one.

500 At this point, all the pictures have been read in from disk and saved in various parts of extended memory.

530 Just initialization. See lines 600 through 630.

570 Remember that a **WHILE** loop executes so long as the expression following **WHILE** is true. But a constant other than zero is always true. So we loop until the user hits BREAK or RESET.

600-620 This is a little sneaky. Every time we get to line 600, **Pic** will be equal to **Oldpic**, so the **WHILE** loop will execute at least once. BASIC XE's **RANDOM** function conveniently chooses a valid picture number. Then we go back up to the top of the **WHILE** loop to find out if we picked a different picture. If not, we try again.

630 And this ensures that the loop of lines 600 to 620 will execute at least once next time.

670-700 Does this code look almost the same as that in lines 440 to 470? It should. The only difference is that now we are moving from the extended memory into the screen memory.

740 As long as you hold the Start key down, BASIC XE will loop on this line. Remember, the "&" symbol means "bit-wise AND," so the test here is of a single bit in the console register.

750 The end of the "forever" loop.

Finally, a last hint of another direction to explore. Although this program used **BGET** to move a picture into screen memory and then **MOVE**d the picture into extended memory, you can also use **BGET** to read directly into extended memory. It won't look as pretty as the files are being read in, but you could remove line 400 and change line 470 to read as follows:

```
470 Bget #1,Address,Picsize,Bank
```

The fast slide show portion of the program is unaffected, because the pictures are still in the memory locations where it expects them. And, if you hit Break but want to continue the show, just type in the following line:

```
GRAPHICS 24:GOTO 500
```

to use the default colors. Or add **SETCOLOR**s before the **GOTO** if you wish.

SHOWPIC Program

```
100 Rem *************
110 Rem *           *
120 Rem *  SHOWPIC  *
130 Rem *           *
140 Rem *************
150 Rem
160 Rem set up buffers, arrays, constants
170 Rem
180 Dim Files$(8,20),File$(20)
190 Picsize=40*192
200 Console=$d01f:Start=$01
210 Rem
220 Rem find all the pictures files
230 Rem
240 Open #1,6,0,"D1:*.PIC"
250 For Pic=1 To 8
260 Input #1,File$
270 If File$(2,2)<>" " Then Pop:Goto 300
280   Files$(Pic;)="D1:",File$(3,10)," "
290   Files$(Pic;Find(Files$(Pic;)," ",0))=".PIC"
300 Next Pic
310 Maxpic=Pic-1
320 Close #1
330 Rem
340 Rem read in all the files
350 Rem
360 Graphics 24
370 Setcolor 2,6,8:Setcolor 4,6,8:Setcolor 1,6,8
380 For Pic=1 To Maxpic
390   Open #1,4,0,Files$(Pic;)
400   Bget #1,Dpeek($58),Picsize
410   Rem
420   Rem move picture into extended memory
430   Rem
440   Bank=Int((Pic-1)/2)
450   Address=$4000
460   If Pic&1=0 Then Address=$6000
470   Move Dpeek($58),Address,Picsize,Bank
480   Close #1
490 Next Pic
500 Rem
510 Rem now show the pictures
520 Rem
530 Oldpic=0:Pic=0
540 Rem
550 Rem we want to do this forever
560 Rem
570 While 1
580   Rem be sure we don't show same one
590   Rem .                  twice in a row
600   While Pic=Oldpic
610     Pic=Random(1,Maxpic)
620   Endwhile
630   Oldpic=Pic
640   Rem
650   Rem move from extended memory to screen
660   Rem
670   Bank=Int((Pic-1)/2)
680   Address=$4000
690   If Pic&1=0 Then Address=$6000
700   Move Address,Dpeek($58),Picsize,Bank
710   Rem
720   Rem allow user to look at one
730   Rem
740   While Peek(Console)&Start=0:Endwhile
750 Endwhile
```

Space For Your Notes

Error Situations

Whenever something that BASIC XE wasn't expecting happens, BASIC XE will stop whatever it's doing and give an error (unless, of course, you **TRAP** the error). An explanatory message will accompany the error number if you have booted with the extensions disk, otherwise the error number alone will be displayed. All errors that involve BASIC XE directly have personalized error messages, but some obscure system errors simply produce the message "(See Manual)". This are errors like #173 (can't format disk), and occur very rarely. The "(See Manual)" does not necessarily mean this manual, but the manual for the device or subsystem that produces the error.

Error	Screen Message and Further Description
1	**BREAK key not TRAPped** While **SET 0,1** was specified, the user hit the <BREAK> key. This **TRAP**able error gives the BASIC XE programmer total system control.
2	**Memory Full** You have used all available memory. You can't enter any more statements, nor can you define any more variables.
3	**Value Out of Range** An expression or variable evaluates to an incorrect value. For example, if a value 0-7 is required, and you use a negative number or a number greater than 7, an error 3 will occur (e.g., **SETC. 99,0,0**).
4	**Too Many Variables** No more variables can be defined. The maximum number of variables is 128.
5	**Access Past String DIM** You tried to access a character beyond the **DIM**ensioned length of a string.
6	**No DATA to READ** A **READ** statement is executed after the last adata item in the last **DATA** statement has already been read.
7	**Val > 32767** BASIC XE encountered a line number larger than 32767. Some other commands (e.g., **POINT**) can also produce this error.
8	**INPUT/READ Type Mismatch** The **INPUT** or **READ** statement did not receive the type of data (arithmetic or string) it expected.
9	**DIMensioning** Either you tried to re**DIM**ension an already **DIM**ensioned var, or used an un**DIM**ensioned variable as though it were **DIM**ensioned.

Error	Screen Message and Further Description
10	**Expression too Complex** An expression is too complex for BASIC XE to handle. The solution is to break the calculation into two or more BASIC XE statements.
11	**Overflow/UnderFlow** The floating point routines have produced a number that is either too large or too small.
12	**Line Not Found** The target lineno of a GOTO, GOSUB, or IF/THEN does not exist.
13	**NEXT without FOR** A NEXT avar was encountered without a corresponding FOR avar. **Note:** Improper use of POP could cause this error.
14	**Line Too Long or Complex** The progam line just entered is either longer or more complex than BASIC XE can handle. The solution is to break the line into multiple lines by putting fewer statements on a line, or by evaluating the expression in multiple statements.
15	**Line Not Found** The line containing a GOSUB or FOR was deleted after it was executed but before the RETURN or NEXT was executed. This can happen if, while running a program, a STOP is executed after the GOSUB or FOR, then the line containing the statement is deleted, then you type CONT and the program tries to execute the RETURN or NEXT.
16	**RETURN without GOSUB** A RETURN was encountered when execution is not in a GOSUB routine. **Note:** improper use of POP could also cause this error.
17	**Bad Line** You tried to RUN a program that had a line with an already-marked syntax error on it (i.e. it has the "ERROR -" on it). Note: the SAVEing of a line that contains a syntax error can be useful when debugging your program, but don't forget to change it before RUNning again.
18	**Not a Number** If the sexp in a VAL does not start with a number, this message number is generated. For example, VAL("ABC") would cause this error.
19	**Too Big to LOAD** The program you're trying to LOAD is larger than available memory. This could happen if you have used LOMEM to change the address at which the BASIC XE tables start, or if you're LOADing using a DOS different from the one used when the program was SAVEd.
20	**Invalid Channel #** If the device number given in an I/O statement is greater than 7 or less than 0, then this error is issued.

Error	Screen Message and Further Description
21	**File Not LOAD format** This error results if you try to **LOAD** a file that was not created by **SAVE.**
22	**USING String Too Big** This error occurs if the entire format string in a **PRINT USING** statement is longer than 255 characters. It also occurs if a single format field is longer than 59 characters.
23	**USING Value Too Big** The value of an aexp in a **PRINT USING** statement is greater than or equal to 1E+50.
24	**USING Type Mismatch** The format field in a **PRINT USING** statement and the corresponding exp to be output using that format are not of the same data type (arithmetic or string).
25	**RGET DIM Mismatch** A string being retrieved by **RGET** has a different **DIM**ensioned length than the string variable to which it is to be assigned.
26	**RGET Type Mismatch** The record element being retrieved by **RGET** and the variable to which it is assigned are not of the same data type.
28	**Invalid Structure** The end of a control structure like **ENDIF** or **ENDWHILE** was encountered without a corresponding **IF** or **WHILE.**
29	**P/M # Out of Range** An illegal player/missile number. Players must be numbered from 0-3 and missiles from 4-7.
30	**P/M Graphics not Active** You attempted to use a **PMG** statement before initializing P/M's via **PMG. 1** or **PMG. 2.**
32	**ENTER not TRAPped** End of **ENTER.** This is the error resulting from using a **SET 9,1.**
34	**Can't NUM/RENUM** aexp1 or aexp2 in a **RENUM** or **NUM** statement evaluated to zero.
35	**Can't NUM/RENUM** When **RENUM**bering, the maximum line number (32767) was exceeded.
40	**String Type Mismatch** You attempted to use an svar as an savar, or visa versa.

Error	Screen Message and Further Description
65	**EXTENDed Memory Not Available** You tried to **LOAD** an **EXTENDed** program or use the **EXTEND** statement on a computer that doesn't have extended memory.
100	**Extensions not installed!** You used a command avialable only if you boot with the disk extensions. See How to Boot BASIC XE in the introduction for a list of these commands.
129	**Channel Already OPEN** You are trying to **OPEN** a CIO channel that is already **OPEN.**
130	**No Device Handler** CIO could not find the device you specified in its device table.
131	**Write Only** You are trying to read from a CIO channel that was **OPENed** for writing only.
132	**Bad Device Cmd** The I/O command you issued does not exist for the device. This can happen if your **XIO** command or **OPEN** mode is wrong.
133	**Channel Not OPEN** You tried to use a CIO channel that you haven't yet **OPENed.**
135	**Read Only** You are trying to write to a CIO channel that was **OPENed** for reading only.
136	**End-Of-File** There is no more data in the file you are reading.
138	**Device Timeout** The device you tried to access did not respond within its allotted time.
139	**Device NAK** The device does not acknowledge.
141	**Screen Position** You tried to access a position not valid in the current graphics mode.
144	**Device Done** Either the I/O operation you attempted didn't execute properly, or you tried to write to a write-protected disk.
145	**Invalid GR Mode** You attempted to use a graphics mode that doesn't exist.
147	**No Memory for GR Mode** You don't have enough room for the graphics mode you specified.

Error	Screen Message and Further Description
160	**Invalid Drive #** DOS does not recognize the drive number you gave. This can happen if you specified an illegal drive number or if the drive is not on.
161	**Too Many OPEN Files** DOS does not have any more buffers available on which to OPEN files.
162	**Disk Full** There is no room for more data on the disk.
165	**Bad File Name** You used an illegal disk file name. See your DOS manual for legal file names.
167	**File PROTECTed** You tried to write to a PROTECTed file.
169	**DIRectory Full** The disk directory is full, so you can't create any new files.
170	**File Not Found** DOS can't find the file you specified on the disk.
171	**Bad Point Value** You attempted to POINT to a non-existent place on the disk, or you did not OPEN the file in update mode (12).

Space For Your Notes

INDEX

Underlined page numbers refer to sections where the term is defined.

BASIC XE™

Just look at what you get for one low sticker price:

BEST MILEAGE: With over 60,000 **more** bytes for your programs, BASIC XE lets you use all the memory you paid for.*

MORE HORSEPOWER: Run Atari BASIC programs 2 to 6 times faster.* Even with its incredible power, BASIC XE is compatible with Atari BASIC.

CLASSIC DESIGN: Show off the sleek **structured** style of your own programs when you use BASIC XE statements like PROCEDURE, IF...ELSE, and WHILE...ENDWHILE.

FREE ACCESSORIES: Get over $100 worth of Atari BASIC options FREE when you buy BASIC XE: complete Player/Missile Graphics support, string arrays, DOS access, SORT commands, readable listings...over 50 extras at no additional charge.