BASIC XL

Compatible!

Flexible!

Fast!

OSS **Precision**
Software Tools™

A REFERENCE MANUAL

For

BASIC XL

This book is Copyright © 1983 by
Optimized Systems Software, Inc.
1221-B Kentwood Avenue.
San Jose, CA 95129

Portions of this book are
Copyright © 1980 Atari, Inc.
And are reprinted with the
permission of Atari, Inc.

Reprinted by:

8-Bit Classics
http://www.8bitclassics.com

NOTE: Minor corrections, changes, and layout were made when reprinted.
This should not affect the manual in anyway.

ACKNOWLEDGEMENT

CAVEAT

TRADEMARKS

Table of Contents

PROGRAM CONTROL STATEMENTS
CHAPTER 4

INPUT / OUTPUT COMMANDS AND DEVICES
CHAPTER 5

SCREEN GRAPHICS AND SOUND
CHAPTER 7   109

1.1 Features of BASIC XL

Compatibility with Atari BASIC

Because BASIC XL uses the same tokens as Atari BASIC, programs written in Atari BASIC which have been SAVEd can be LOADed and RUN using BASIC XL.

FAST Program Execution

BASIC XL allows you to RUN your programs faster than ever with the new FAST command, thus making games written in BASIC almost as fast as arcade games.

Easy Program Formatting

Unlike other BASICs, BASIC XL does not care whether you use upper or lower case letters when you enter your programs. This alone makes programs more readable. However, BASIC XL does even more. 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, so your programs are easier to follow, no matter how complex or involved they are.

Built-in Functions

BASIC XL contains over 40 built-in functions covering a wide range of applications. The chapter titled FUNCTION LIBRARY explains these functions and their usages.

Graphics

BASIC XL offers the same bit-map graphics manipulation available in Atari BASIC, and allows amazing flexibility in color choice and pattern variety. Chapter 7 explains each command and give examples of the many ways to use each.

Player / Missile Graphics

BASIC XL allows you easy access to the player / missile graphics available on the Atari through the use of built-in functions and commands. With BASIC XL, p/m graphics are as easy to control as common bit-map graphics.

Game Controllers

Not only does BASIC XL support the game controller functions as Atari BASIC, but it also adds some other game controller functions which make interpreting and using the joysticks much easier.

Sound

The Atari Personal Computer is capable of emitting a large variety of sounds including simulated explosions, electronic music, and "raspberries", and BASIC XL allows you to have control over these sounds available.

Wraparound and Keyboard Repeat

If you enter a program line which is longer than the length of the screen, the line "wraps around" to the next line so that you can view it. Also, if you hold down any key for over ½ second, it will start repeating.

Error Messages

If a data entry error is made, the screen display shows an error message and the line on which the error occurred (with the character at which the error occurred highlighted). Most errors will also display a short, descriptive message along with the error number. Appendix A contains a list of all the error messages and their explanations.

1.2 Special Notations used in this Manual

Line Format

The format of a line in a BASIC program includes a line number (abbreviated to lineno) at the beginning of the line, followed by a statement keyword, followed by the body of the statement and ending with a line terminator command (<RETURN> key). In an actual program, the four elements might look like this:

lineno	Statement Keyword	Statement Body	Terminator
100	PRINT	A/X*(Z+4.567)	<RETURN>

Several statements can be typed on the same line provided they are separated by a colon (:).

Capital Letters

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

Lower Case Letters

In this manual, lower case letter are used to denote the various classes of items which may be used in a program, such as variables (var), expressions (exp), and the like.

<u>Items in Brackets</u>

Brackets ([ ]) contain optional items which may be used, but are not required, in the format of a statement. If the item enclosed in brackets is followed by three dots (e.g. [exp, …]), more than one of that item may be entered, but none are required.

<u>Items Stacked Vertically in Bars</u>

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 example below, type either the GOTO or the GOSUB

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

<u>Command abbreviations in headings</u>

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

1.3 Glossary and Terminology

adata	(ATASCII Data) Any ATASCII character, excluding commas and carriage returns. (See Appendix C.)
aexp	(Arithmetic Expression) General composed of a variable, functions, constant, or two arithmetic expressions separated by an arithmetic operator. See section 2.3.2.
alphanumeric	The letters A through Z (either lower or upper case) and the digits 0 through 9.
aop	(Arithmetic operator). See section 2.2.1.
Arrays and Array Variables	An array is a list of places where data can be filed for future use. Each of these places is called an element, and the whole array or any element is called an array variable. See section 2.1.2.
avar	(Arithmetic Variable) A location where a numeric value is stored. Variable names may be from 1 to 120 alphanumeric characters, but must start with an alphabetic character. All characters are normalized to upper case normal (i.e., not inverse) video.
BASIC	Beginner's All-purpose Symbolic Instruction Code.
Constant	A constant is a value expressed as a number rather than represented by variable name. For example, in the statement X = 100, X is a variable and 100 is a constant.
Command String	Multiple commands (or program statements) placed on the same numbered line separated by colons.

exp	Any expression, whether sexp or aexp. See section 2.3.
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 (See aexp and sexp).
filespec	File Specification: A string expression that refers to a device such as the keyboard or to a disk file. It contains information on the type of I/O device, its number, a colon, an optional file name, and an optional filename extender. See section 5.2.

NOTE: BASIC XL allows you to omit the double quotes normally required in a literal string when the literal string is used as a filespec for any of the following commands:

    DIR   LOAD   PROTECT   LVAR   RUN
    ENTER SAVE   RENAME   OPEN   XIO

CAUTION: When filespec is used this way, it must be the last thing on the program or command line. Also, DIR, LVAR, and RUN must always be the last command on the line. |
Function	A function is a subroutine built into the computer so that it can be called by the user's program. A function is NOT a statement. COS (Cosine), FRE (unused memory space), and INT (integer) are examples of functions. In many cases the value is simply assigned to a variable (stored in a variable) for later use. In other cases it may be printed out on the screen immediately. See chapter 5 for more on functions.
Keyword	Any reserved word "legal" in the BASIC language. May be used in a statement, as a command, or for any other purpose. (See Appendix A for a list of all "reserved words" or keywords in BASIC XL.)
lineno	(Line Number) A constant that identifies a particular program line in a deferred more BASIC program. Must be an integer from 0 through 32767. Line numbering determines the order of program execution.
Logical Line	A logical line consists of one to three physical lines, and is terminated either by a <RETURN> or when the maximum logical line limit is reached. Each numbered line in a BASIC program consists of one logical line when displayed on the screen.
lop	(Logical Operator) See section 2.2.2.

mvar	(Matrix Variable) Also called a Subscripted Variable. An element of an array or matrix. The variable name for the array or matrix as a whole may be any legal variable name. See section 2.1.2.
Operator	Operators are used in expressions to tell the computer how it should evaluate the variables, constants, and functions in the expression. There are two types of operators – arithmetic and logical. For more information, see section 2.2.
Physical Line	One line of characters as displayed on a TV or monitor screen.
sexp	(String Expression) Can consist of a string variable, string literal (constant), or a function that returns a string value. See section 2.3.3.
String	A string is a group of characters enclosed in quotation marks. "ABRACADABRA" is a string. So are "OSS IS THE BEST" and "123456789". A string is much like a numeric constant (e.g., 12.4), as it may be stored in a variable. A string variable is different in that its name must end in the character $. See section 2.1.3.
svar	(String Variable) A location where a string of characters may be stored. See 2.1.3 and 2.1.4.
var	(Variable) Any variable. May be mvar, avar, or svar. See section 2.1.
Variable	A variable is the name for a numerical other quantity which may (or may not) change. Variable names may be up to 120 characters long. However, a variable name must start with an alphabetic letter, and may contain only letters and digits. See section 2.1.

1.4 Operating Modes

Direct Mode

Uses no line numbers and executes instruction immediately after <RETURN> key is pressed.

Deferred Mode

Uses line numbers and delays execution of instruction(s) until the RUN command is entered.

Execute Mode

Sometimes called RUN mode. After the RUN command is entered, each program line is processed and executed.

Memo Pad Mode

A non-programmable mode that allows the user to experiment with the keyboard or to leave messages on the screen. Nothing written while in Memo Pad mode affects the RAM-resident program.

NOTE: this mode is only available on the Atari 400 and 800.

2.1 Variables (var)

There are two basic types of variables in BASIC XL – arithmetic variables and string variables. Also, there are three extensions to these – arrays, matrices, and string arrays.

Arithmetic, array, and matrix variables all store numbers, and can only be used where a number is required.

String and string array variables both store character strings and can only be used where a character string is required.

There are limits to the number of variables you may use, and to the size and format of a variable name, as follows:

> 1) BASIC XL limits the user to 128 variable names. To bypass this problem, use individual elements of any array instead of having separate variable names. To clear the variable name table (possibly after an error 4), you can save your program using LIST, then type NEW, and then ENTER your program back in.

> 2) All variable names must start with an alphabetic letter, followed by either letters or digits. The name must be less that 120 characters long. All string or string array variable names must end in the 'S' (dollar sign) character.

2.1.1 Arithmetic Variables (avar)

Arithmetic variables are those which store a single number, and are the most common variables used. The following are examples of arithmetic variables.

```
X
THISISANARITHMETICVARIABLE
TEMP
CHARGE
```

Here are some examples of arithmetic variables in use:

```
100 LET X=76        : REM here's on use
200 FOR I=1 TO 100 : REM here's a second
300 PRINT X-2       : REM and a third
400 NEXT I
500 END
```

2.1.2 Array / Matrix Variables (mvar)

An array variable is a group of memory location (called elements or subscripts of the array). In each one of these locations is a number; so, in essence, an array is simply a group of arithmetic variables which share a common name.

The manner in which you access a given element of an array is simple – you merely give the array name followed by the element number in parentheses, as in the following examples:

    A(3)   ARRAY(14)   NUMLIST(40)

The elements are numbered starting at 0, and continue through to the DIMensioned size of the array. "How do I dimension the size?" It's easy. You use the DIM statement as follows:

    DIM A(40)   REM dimension 'A' as a 40 element
                         REM array.
    DIM NUMLIST(60)   REM dimension 'NUMLIST' as a 60
                         REM element array.

For more information on the use of DIM, see section 2.1.5.

A matrix is similar to an 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. You can think of a matrix as a grid, with each box being one element. The following is a representation of a 5 by 5 matrix, where each of the boxes contains the subscripts used to access that box (element):

Notice that the row number is given first, followed by a comma and then the column number. This is the same order you would use to access that element.

		C	O	L	U	M	N
R		0,0	0,1	0,2	0,3	0,4
O		1,0	1,1	1,2	1,3	1,4
		2,0	2,1	2,2	2,3	2,4
W		3,0	3,1	3,2	3,3	3,4
		4,0	4,1	4,2	4,3	4,4

Dimensioning the size of a matrix is very similar to dimensioning an array, but both the row dimension and column dimension are required, e.g.:

    DIM AMATRIX(4,4)   REM a 5 by 5 matrix; remember
                             REM that (0,0), not (1,1) is
                             REM the first element.

NOTE: for more information on DIM, see section 2.1.5.

When you use an element of an array or matrix, you are actually using a single number (which is what an arithmetic variable is). This means that an array or matrix element may be used wherever 'avar' can be used.

Examples:

    X=47.4
    ARRAY(7)=47.4
    MATRIX(4,3)=47.4

    IF ABS(X) < 100 THEN …
    IF ABS(ARRAY(7)) < 100 THEN …
    IF ABS(MATRIX(4,3)) < 100 THEN …

2.1.3 String Variables (svar)

String variables are used to store literal strings of characters. A literal string of characters enclosed in double quotes:

    "this is a literal string"
    "numbers in quotes are strings: 34344.2"

String variable names are just like arithmetic variable names, except that they must end with a '$', as in the following examples:

    STRING$
    A$

To dimension the size of a string variable (i.e., define how many characters it may hold), you use the DIM statement (also see 2.1.5):

    DIM STRING$(66)
    DIM A$(10)

NOTE: BASIC XL will auto-dimension a string variable if you don't manually DIMension it. See 3.15 for more info on this feature.

With arrays and matrices the first element is the zeroeth, but with strings the first element is the first, e.g.:

    DIM A$(10)
    A$="A String"

2.1.4 String Array Variables (svar)

A string array is very similar to a normal arithmetic array (section 2.1.2), except that each element is a string, not a number.

As with string variables, a string array variable must have its name end with a '$', and it is dimensioned using DIM. However, there are two quantities which need to be dimensioned – the number of elements and the size of each element. The following examples show how to do this (also see section 2.1.5):

>   DIM Strarray$(4,40)
>   DIM A$(10,100)

The first example dimensions a string array called "Strarray$" with 4 elements. Each element is a string 40 characters long. The second example dimensions the string array "A$" to 10 elements, with each element being 100 characters in length.

To access one of the elements of a string array you specify the element number (the first element is a number 1, not 0 as in arithmetic arrays) followed by a semicolon (;). An example follows:

>   100 DIM A$(3,6)
>   200 A$(1;)="TEST"
>   300 A$(2;)="STRING"
>   400 A$(3;)="ARRAY"

2.1.5 DIM

Format: DIM svar(aexp[,aexp]) [,svar(aexp[,aexp])...]
          DIM mvar(aexp[,aexp]) [,mvar(aexp[,aexp])...]

Example: DIM A(100)
          DIM M(6,3)
          DIM B$(20)
          DIM A$(20,40)

A DIM statement is used to reserve a certain number of locations in memory for an array, matrix, string, or string array.

The first example reserves 101 locations (each of which can contain any legal numeric quantity) for an array designated A.

The second example reserves 7 rows by 4 columns for a two-dimensional array (matrix) designated M.

The third example reserves 20 bytes for the string 'B$'.

NOTE: BASIC XL contains an auto DIMension capability for simple string variables only which you can control. For more info, see SET, section 3.15.

The fourth example reserves a string array of 20 elements, with each string element being 40 characters long.

2.2 Operators

BASIC XL has two types of operators:

1) Arithmetic Operators
2) Logical (relational) Operators

As you will see in the expressions sections, either of these two types of operators may be used in arithmetic expressions, while neither may be used in a string expression.

2.2.1 Arithmetic Operators (aop)

BASIC XL uses 7 arithmetic operators:

+	addition (also unary plus; e.g., +5)
-	subtraction (also unary minus; e.g., -5)
*	multiplication
/	division
^	exponentiation
&	bitwise "AND" of two positive integers (both <= 65535)
!	bitwise "OR" of two positive integers (both <= 65535)
%	bitwise "EOR" of two positive integers (both <= 65535)

The first four are straightforward enough, but the last four require some explanation.

The "^" operator is used to raise a number to a specified power. The following examples should clarify this:

Exponent	Expanded	Result
4^2	4*4	16
5^3	5*5*5	125

& tests two bytes bit by bit, returning a value based on this table:

	Bit A	Bit B	Bit-wise And Result
	1	1	1
	0	1	0
	0	0	0
	1	0	0

Example: 5 & 39 --   00000101 (equals 5 decimal)
00100111 (equals 39 decimal)
&   ------------
00000101 (result of & is 5)

	Bit A	Bit B	Bit-wise And Result
! returns a value dependent on this table:	1	1	1
	0	1	1
	0	0	0
	1	0	1

```
Example: 5 ! 39 --   00000101 (5)
                     00100111 (39)
                 !   ------------
                     00100111 (result of ! is 39)
```

	Bit A	Bit B	Bit-wise And Result
% returns a value dependent on this table:	1	1	0
	0	1	1
	1	0	1
	0	0	0

```
Example: 5 % 39 --   00000101 (5)
                     00100111 (39)
                 %   ------------
                     00100010 (result of % is 34)
```

2.2.2 Logical Operators (lop)

The logical operators consist of three types: relational, unary, and binary.

The rest of the binary operators are relational.

<	The first expression is less than the second expression.
>	The first expression is great than the second.
=	The expressions are equal to each other.
<=	The first expression is less than or equal to the second.
>=	The first expression is greater than or equal to the second.
<>	The two expressions are not equal to each other.

Examples:

$$X >= 7$$
$$X <> INT(Y)$$

These operators are most frequently used in IF/THEN statements (i.e., in relational tests), but may also be used in arithmetic expressions. When used in this way, a 1 results the logical test proved true, and a 0 results if the test proved false.

The unary operator is NOT, and the binary operators are:

AND -- Logical AND
OR -- Logical OR

Examples:

10 IF A=12 AND T=0 THEN PRINT "GOOD"	Both expressions must be true before GOOD is printed (that is, A must equal 12 and T must equal 0).
10 A=(C>1) AND (N<1)	If both expressions true, A = +1; otherwise A = 0.
10 A=(C+1) OR (N-1)	If either expression true, A = +1; otherwise A = 0.
10 A=NOT(C+1)	If expression is false, A=+1; otherwise A = 0.

2.2.3 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? Without operator precedence it's impossible to tell. BASIC XL'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
< > = <= >= <>	Relational Operators when used to evaluate strings in arithmetic expressions
NOT + -	NOT, Unary Plus and Minus
^	Exponentiation
% ! &	Bitwise EOR, OR, AND
* /	Multiplicative Operations
+ -	Additive Operators
< > = <= >= <>	Relational Operators
AND	Logical 'and'
OR	Logical 'or'

Examples showing the above precedence in use can be found in section 2.3.2.

2.3 Expressions (exp)

Expressions are constructions which obtain values from variables, constants, and functions using a specific set of operators. In BASIC XL there are two types of expressions -- arithmetic and string. Each of these is dealt with separately, but before going into the expressions themselves something needs to be said about the constant numbers used in arithmetic expressions.

2.3.1 Numbers

All numbers in BASIC XL are BCD floating point, but there are two ways to enter them -- in decimal or hexadecimal.

Decimal numbers may either be whole integers, fractions, or scientific notation. The following are examples of each:

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 multiplied by 10 squared").

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

$4A30	-$0A	$6FF
-$E	-$A000	$FFFF

The maximum hexadecimal value allowed is $FFFF.

Internal Format of Numbers:

Numbers are represented internally in 6 bytes. There is a 5 byte mantissa containing 10 BCD digits and a one byte exponent.

The most significant bit of the exponent byte gives the sign of the mantissa (0 for positive, 1 for negative). The least significant 7 bits of the exponent byte gives the exponent in excess 64 notation. Internally, the exponent represents powers of 100 (not powers of 10).

Example:

$$0.02 = 2 * 10^{-2} = 2 * 100^{-1}$$

exponent $\qquad$ = -1 + 40 = 3F

0.02 $\qquad$ = 3F 02 00 00 00 00

The implied decimal point is always to the right of the first byte. An exponent less than hex 40 indicates a number less than 1. An exponent greater than or equal to hex 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 significant when INPUTing a number. Internally the user can usually get 10 significant digits in the special case where there are an even number of digits to the right of the decimal point (0,2,4…).

2.3.2 Arithmetic Expressions (aexp)

Arithmetic expressions are those which evaluate to a number. Following is a list of expression elements which are considered to be numbers:

      1)   a constant number
      2)   an avar (including subscripted mvars)
      3)   a function which returns a number
      4)   two sexps compared using a relational operator.

The first three are straightforward, but the fourth requires an example:

        100 S$=”ABC”
        200 PRINT S$< “DEF”
        300 END

prints out:

        1

because the logical comparison of the two strings is true.

An arithmetic expression can simply be one of the above, or two or more of the above separated by operators (either arithmetic or logical). The following are examples of arithmetic expressions, including the order of the operators' evaluation (in any) and the result:

Expression	Evaluation Order	Result
3*(4+(21/7)*2)	/,*,+,*	30
“ABC”>”DEF”+7*(ASC(“A”))	>,ASC,*,+	455
X=100 : Y=2		
INT (X*Y/3)	*,/,INT	66

2.3.3 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 expression possibilities:

- 1) a string constant
- 2) an svar (including subscripted string arrays)
- 3) a function which returns a string
- 4) a substring of an svar or string array

This is the first time we've seen the word "substring" used, so we need to define and to explain it.

String	Definition when Destination String	Definition when Source String
S$	the entire string 1 thru DIM value	from 1st thru LEN character
S$(n)	from nth thru DIMth character	from nth thru LENgth character
S$(n,m)	from the nth thru the mth character	from the nth thru the mth character
SA$(e;)	same as S$, except string eth element of SA$	same as S$, except string is eth element of SA$
SA$(e;n)	same as S$, except string is eth element of SA$	same as S$(n), except string is eth element of SA$
SA$(e;n,m)	same as S$(n,m) except string is eth element of SA$	same as S$(n,m) except string is eth element of SA$

A destination string is one to which something is being assigned. Any other string is a source string. In

X$=Y$	READ X$	INPUT X$
RPUT Y$	PRINT Y$	etc.

X$ is the destination string, Y$ is the source string.

An error occurs if either the first or last specified character (n and m, above), or the element number (in the case of string arrays) is outside the DIMensioned size. Also, an error occurs if the last character position given (explicitly or implicity) is less than the first character position.

Source Example: (Assume A$ = "VWXYZ")

        1) PRINT A$(2)   prints: WXYZ
        2) PRINT A$(3,4)   prints: XY
        3) PRINT A$(5,5)   prints: Z
        4) PRINT A$(7)
           is an error because A$ has a length of 5.

Destination Example: (Assume DATA "VWXYZ")
        1)   READ D$
             PRINT D$ prints: VWXYZ

Some of the commands available in BASIC XL are designed specifically to aid in quick and effective program development. The operations these commands execute are too diverse to describe in detail here, so we'll simply give their names and refer you to the section in which the particular command is discussed:

BYE	LIST	RENUM
CLAR	LOMEM	RUN
CONT	LVAR	SET
DEL	NEW	STOP
DOS	NUM	TRACE
FAST	REM	TRACEOFF

3.1 BYE (B.)

      Format: BYE

      Example: BYE

The function of the BYE command is to exit BASIC XL and put the computer in Memo Pad mode. This allows you to experiment with the keyboard or to leave messages on the screen without disturbing any BASIC XL program in memory. To return to BASIC XL press <SYSTEM RESET>.

3.2 CLR

      Format: CLR

      Example: 200 CLR

This command clears the memory of all previously dimensioned strings, arrays, and matrices so the memory and variable names can be used for other purposes. It also clears the values stores in undimensioned variables. If a matrix, string, or array is needed after a CLR command, it must be redimensioned with a DIM command.

3.3 CONT (CON.)

Format: CONT

Example: CONT
          100 CONT

In direct mode, this command resumes program execution after a STOP statement, a <BREAK> key abort, or any stop caused by an error.

CAUTION: Execution resumes on the line following the halt, so any statements following the halt (and on the same line as the halt) will not be executed.

In deferred mode, CONT may be used for error trap handling.

Example:
          10 TRAP 100
          20 OPEN #1, 12, 0, "D:X"
          30
          ..
          ..
          100 IF ERR(0)=170 THEN
              OPEN #1,8,0, "D:X":CONT

In line 20 we attempt to open a file for updating. If the file does not exist, a trap to line 100 occurs. If the "FILE NOT FOUND" error occurred, the file is opened for output (and thus created) and execution continues at line 30 via "CONT".

3.4 DEL

Format: DEL line[,line]

Example: DEL 1000,1999

DEL deletes program lines currently in memory. If two line numbers are given (as in the example), all lines between the two numbers (inclusive) are deleted. A single line number deletes a single line.

Examples:
          100 DEL 1000,1999
          110 SET 9,1:TRAP 1000
          120 ENTER "D:OVERLAY1"
          1000 REM These lines are deleted by line 100.
          1010 REM Presumably they will be overlaid by
          1998 REM the program ENTERed in line 120.
          1999 REM See 'ENTER' and 'SET' for more info.

3.5 DOS

>   Format: DOS
>
>   Example: DOS

The DOS command is used to go from BASIC XL to the Disk Operating System (DOS). If the Disk Operating System has not been booted into memory, the computer will go into Memo Pad mode and the user must press <SYSTEM RESET> to return to Direct Mode. If the Disk Operating System has been booted, control is given to DOS. To return to BASIC XL, press 'CAR' <RETURN> for OS/A+ or DOS XL, or press 'B' <RETURN> for Atari DOS.

NOTE: The command CP is exactly equivalent to DOS.

DOS is usually used in Direct mode; however, it may be used in a program. For more details on this, see your DOS manual.

3.6 FAST

>   Format: [lineno] FAST
>
>   Example: FAST
>   100 FAST

During normal program execution BASIC XL must search (from the beginning) for a specified line number whenever it encounters a GOTO, GOSUB, FOR, or WHILE (this is how most of the other BASICs do it too). However, you can change this by using the FAST command.

When BASIC XL sees 'FAST', it does a precompile of the program currently in memory. During the precompile BASIC XL changes every line number to the address of that line in memory. Now, when a GOTO, GOSUB, FOR, or WHILE is executed, no line number search is needed, since BASIC XL can simply jump right 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), then that lineno will not be affected by FAST, and so will RUN at normal speed.

3.7 LIST (L.)

        Format: LIST [lineno [, lineno]
                              LIST ["filespec"[,lineno [, lineno]]]

        Example: LIST
                    LIST 10
                    LIST 10, 100
                    LIST 10,
                    LIST "P:"
                    LIST "D:DEMO.LST"
                    LIST "P:", 20, 100

LIST causes the program currently in memory to be displayed. You can display a single line by giving the line number after the 'LIST', or display a group of lines by giving the starting line number and ending line number (separated by a comma) after the 'LIST'.

If you give the starting line number, a comma, and no end address, the ending line number is assumed to be the last line in the program.

If no line number(s) is given, the entire program is displayed.

You can also redirect the display to a file by entering the filespec enclosed in double quotes immediately after the 'LIST'. You can then add any of the line number specifications described above to list only what you want to that file.

LIST can be used in Deferred mode as part of an error trapping routine (See TRAP in Section 4).

NOTE: the quotes around the filespec are required for LIST, unless of course a string variable is used.

3.8 LOMEM

        Format: LOMEM addr

        Example: LOMEM DPEEK(128)+1024

This command 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.

3.9 LVAR (LV.)

        Format: LVAR [filespec]

        Example: LVAR P:

This statement will list (to any file) all variables currently in use. 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 no filespec is used then LVAR lists to the screen.

NOTE: strings are denoted by a trailing '$', arrays by a trailing '('.

WARNING: LVAR must be the last (or only) command on a line.

3.10 NEW

        Format: NEW

        Example: NEW

This command erases the program stored in RAM. Therefore, before typing NEW, either SAVE or CSAVE any programs to be recovered and used later. NEW clears BASIC's internal symbol table so that no arrays (See Section 8) or strings (See Section 7) are defined. NEW is normally used in Direct mode but is sometimes useful in deferred mode as an alternative to End.

3.11 NUM

        Format: NUM [start][,increment]

        Example: NUM
                      NUM 50
                      NUM ,1
                      NUM 50,1

The NUM command enables BASIC XL's automatic line numbering facility. This facility can increase your program entry speed because it puts in the program line numbers for you.

If no start or increment 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, incrementing by the number you gave it as an increment.

If both the starting line number and the increment are given (last example), NUM will start numbering from the given line number and increment by the given increment value.

Three things cause the automatic line numbering to stop:

    1)   If you press <RETURN> immediately following the line number.

    2)   If a syntax or similar error is encountered on a program line you type in.

    3)   If the next automatic line number is the same as a line number already in the program. This keeps you from overwriting previously written parts of your program.

NOTE: If the starting line number you give already exists, then the automatic line numbering will not begin.

3.12 REM (R.)

Format: REM text

Example: 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 REMark, no statements following it (on the same logical line) will be executed.

3.13 RENUM

> Format: RENUM [start][,increment]

> Example: RENUM
> RENUM 100
> RENUM ,30
> RENUM 1000,5

RENUM renumbers the entire program as it currently resides in memory. The first line in memory is given the line number specified by 'start', and each subsequent line number is one 'increment' greater than the last.

All line number references (e.g., in GOTO, GOSUB, etc.) are also renumbered IF the line numbers are absolute numbers. Line number expression (e.g., GOTO 1000+10*INDEX) will NOT be renumbered.

If no 'start' line number is given, RENUM assumes a starting line number of 10. If no 'increment' is given, RENUM will renumber lines in increments of 10. (That is, just typing 'RENUM' is equivalent to typing 'RENUM 10,10'.)

As noted in the examples above, both start and increment are separately optional.

WARNING: If you use LIST in deferred mode (i.e., in a program) the lineno values you want to list will not be RENUMbered.

WARNING: RENUM will not renumber absolute linenos after a lineno expressed as an expression. Examples:

> ON X GOSUB 100,3*Y,200

In this example 100 will be RNEUMbered, but 200 will not, since it follows a lineno expressed as an expression (3*Y).

3.14 RUN

       Format: RUN [filespec]

       Example: RUN
                   RUN D:MENU

This command causes the computer to begin executing a program. If no filespec is specified, the current RAM-resident program is executed. If a filespec is included, the computer retrieves the tokenized program from the specified file, executes a FAST command (see section 3.6), and then executes the program.

Before execution begins all variables (including arrays, strings, and matrices) are set to zero, all open files (channels) are closed, and all sounds are turned off.

Unless the TRAP command is used, an error will cause the execution to halt and an error message will be displayed.

Run can also be used in Deferred mode.

Example: 10 PRINT "OVER AND OVER AGAIN."
              20 RUN

Type RUN and press <RETURN>. To end, press <BREAK>.

To begin program execution at a point other than the first line number, type GOTO followed by the specific line number, then press <RETURN>. CAUTION: arithmetic variables, arrays, and strings are neither cleared or initialized by GOTO.

NOTE: RUN must be the last (or only) command on a line.

3.15 SET

      Format: SET aexp1,aexp2

      Example: 100 SET 1,5

SET is a statement which allows you to execute control over a variety of BASIC XL system level functions. The table below summarizes the various SET table parameters (default values are given in parentheses).

aexp1		aexp2		Meaning
0	(0)	0	•	BREAK key functions normally
		1	•	User hitting BREAK cause an error to occur (TRAPable)
		128	•	BREAKs are ignored
1	(10)	1 Thru 128	•	Tab stop setting for the comma in PRINT statements.
2	(63)	0 Thru 255	•	Prompt character for INPUT (default is "?").
3	(0)	0	•	FOR…NEXT loops always execute at least once (ala ATARI BASIC).
		1	•	FOR loops may execute zero times (ANSI standard)
4	(0)	0	•	On a multiple variable INPUT, if the user enters to few items, he is reprompted (e.g., with "??")
		1	•	Instead of reprompting, a TRAPable error occurs.
5	(1)	0	•	Lower case and inverse video characters remain unchanged without causing syntax errors (BASIC XL allows mixed case program entry).
		1	•	For program entry ONLY, lower case letter are converted to upper case and inverse video characters are inverted. EXCEPTION: characters between quotes remain unchanged.
				CAUTION: this conversion applies to REMarks and DATA statements also. For total compatibility with ATARI BASIC, it might be best to use SET 5,0.

aexp1		aexp2		Meaning
6	(0)	0	•	Print error messages along with error numbers (for most errors)
		1	•	Print only error numbers
7	(0)	0	•	Missiles (in Player / Missile Graphics), which move vertically to the edge of the screen, roll off the edge and are lost.
		1	•	Missiles wraparound from top to bottom and vice versa.
8	(1)	0	•	Don't push (PHA) the number of parameters to a USR call on the stack [advantage: some assembly language subroutines note expecting parameters may be called by a simple USR(addr)].
		1	•	Do push the count of parameters (ATARI BASIC standard).
9	(0)	0	•	ENTER statements return to the READY prompt level on completion.
		1	•	If a TRAP is properly set, ENTER will execute a GOTO the TRAP line on end-of-entered-file.
10	(0)	0	•	The four missiles act separately; that is as four missiles.
		1	•	The four missiles are grouped into a fifth player, you need only do a PMMOVE of one of the missiles since they are all grouped together.
11	(40)	1 Thru 255	•	BASIC XL will DIM a string to this size if you do not use a DIM statement to otherwise dimension it.
		0	•	BASIC XL works like ATARI BASIC
12	(1)	0	•	The program LIST formatter does not indent when you use structured statement (FOR, WHILE, etc.).
		1	•	The LIST formatter does indent when you use structured statements.

NOTE: The SET parameters are reset to the system defaults on execution of a NEW statement.

Examples:

1) SET 1,4 : PRINT 1,2,3,4

The number will be printed every four columns

2) SET 2, ASC(">")

Changes the INPUT prompt from "?" to ">"

3)   100 SET 9,1 : TRAP 120
      110 ENTER "D:OVERLAY.LIS"
      120 REM execution continues here after
      130 REM entry of the overlay

4)   100 SET 0,1 : TRAP 200
      110 PRINT "HIT BREAK TO CONTINUE"
      120 GOTO 110
      200 REM come here via BREAK KEY

5)   100 SET 3,1
      110 FOR I = 1 TO 0
      120 PRINT "THIS LINE WON'T BE EXECUTED"
      130 NEXT I

3.16 STOP

      Format: STOP

      Example: 100 STOP

When the STOP command is executed in a program, BASIC XL displays the message STOPPED at LINE lineno, terminates program execution, and returns to Direct mode. The STOP command does not close files or turn off sounds (as does END), so the program can be resumed by typing CONT <RETURN> (see section 3.3 for more info on CONT).

3.17 TRACE and TRACEOFF

      Format: TRACE
                  TRACEOFF

      Example: 100 TRACE
                  TRACEOFF

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

Exceptions: The first line of a program does not have its number traced. The object line of a GOTO or GOSUB and the looping line of FOR or WHILE may not be traced.

NOTE: A direct statement (e.g., RUN) is TRACED as having line number 32768.

This chapter explains the commands associated with loops, conditional and unconditional branches, error traps, and subroutines. It also explains the means of accessing data and the optional command used for defining variables.

The following commands are described in this chapter:

Assignment Statement	LET
END	MOVE
FOR…TO…STEP/NEXT	ON…GOTO/GOSUB
GOSUB…RETURN	POP
GOTO	RESTORE
IF…THEN	TRAP
IF…ELSE…ENDIF	WHILE…ENDWHILE

4.1 Assignment Statement

Format: avar=aexp
          mvar(aexp)=aexp
          svar(aexp;)=sexp [,sexp…]
          svar=sexp [,sexp…]

Example: X=9
          I=X+7*9
          ARRAY(7)=23.75
          A$(4;)="A STRING ARRAY ELEMENT"
          S$="THIS IS A STRING"
          M$="CONCENTRATED"
          C$=S$," WHICH IS ",M$

This assignment statement is used to assign a value to a variable, and can be used with arithmetic, matrix (array), or string variables (including string arrays).

The first and second examples given simply equate an avar to an aexp. If you insert a 'PRINT I' statement after the second example, 72 (the value of I) will be printed. The third equates one element of a mvar to an aexp.

The fourth example is somewhat more complicated; it equates on element of a string array to a sexp (in this case a string constant).

The fifth and sixth examples equate svars to sexps.

String concatenation may be accomplished via the form shown in the last example above. Note that

          A$=B$,C$

is exactly equivalent to

          A$=B$
          A$(LEN(A$)+I)=C$

From this you can see that C$ in the last example is equal to "THIS IS A STRING WHICH IS CONCATENATED".

Here is another example:

          100 DIM A$(100),B$(100)
          200 A$="123"
          300 B$="ABC"
          400 A$=A$,B$,A$
          500 REM At this point A$ = "123ABC123ABC"
          600 A$(4,9)="X",STR$(3*7),"X"
          700 REM At this point, A$="123X21X23ABC"

800 A$(7)=A$(1,3)
900 REM Finally, A$="123X21123"

NOTE: for more information on variables and expressions, see chapter 2.

4.2 END

Format: END

Example: 1000 END

This command is used to terminate the execution of a program. In addition to this, it also closes all files and turns off any sounds. It does not change the GRAPHICS mode, however. END is not required in most programs because BASIC XL automatically closes all files and turns off any sounds after the last program line has executed.

If you have any subroutines following the main program you should put an END at the end of the main program; otherwise the subroutines will be executed as part of the main program.

END may also be used in Direct mode to close files and turn off sounds.

4.3 FOR (F.)…TO…STEP / NEXT (N.)

Format: FOR avar = aexp1 TO aexp2 (STEP aexp3)
         NEXT avar

Example: FOR X = 1 TO 10
          NEXT X

          FOR Y = 10 TO 20 STEP 2
          NEXT Y

          FOR INDEX = Z TO 100 * Z
          NEXT INDEX

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 of aexp1. Each time the NEXT avar statement is encountered, the loop variable is incremented by the amount specified by aexp3 in the 'STEP' option. aexp3 can be either positive or negative, either a fraction or a whole number. If the 'STEP' option is not used, the loop increments by one. When the loop completes the limit as defined by aexp2, it stops and the program proceeds to the statement immediately following the NEXT statement.

FOR loops can be nested, one within another. In this case, the innermost loop is completed before returning to the outer loop. The following example illustrates a nested loop program.

```
10 FOR X=1 TO 3 : REM START OF OUTER LOOP
20   PRINT "OUTER LOOP"
30   Z=0
40   Z=Z+2
50   FOR Y=1 TO 5 STEP Z : REM START OF INNER LOOP
60     PRINT "    INNER LOOP"
70   NEXT Y : REM END OF INNER LOOP
80 NEXT X : REM END OF OUTER LOOP
90 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 Z). In this case, Z has been defined as 0, then redefined as Z+2. Using this data, the computer must complete three passes through the inner loop before returning to the outer loop. The following is the output of the program when it is RUN:

```
OUTER LOOP
        INNER LOOP
        INNER LOOP
        INNER LOOP
```

```
OUTER LOOP
        INNER LOOP
        INNER LOOP
        INNER LOOP
OUTER LOOP
        INNER LOOP
        INNER LOOP
        INNER LOOP
```

The return addresses for the loops are placed in a special group of memory addresses referred to as a stack. The information is "pushed" on the stack and when used, the information is "popped" off the stack (see POP).

4.4 GOSUB (GOS.) / RETURN (RET.)

Format: GOSUB lineno1
lineno1
   :
   :
lineno2 RETURN

Example: 100 GOSUB 2000
2000 PRINT "SUBROUTINE"
2010 FOR X=1 TO 10
2020   PRINT X,X*X
2030 NEXT X
2040 RETURN

A subroutine is a program or routine used to compute a certain value, etc. It is generally used when an operation must be executed several times within a program sequence using the same or different values. This command allows the user to "call" the subroutine, if necessary. The last line of the subroutine must contain a RETURN statement. The RETURN statement goes back to the physical line following the GOSUB statement.

Generally, a subroutine can do anything that can be done in a program. It is used to save memory and program-entering time, and to make programs easier to read and debug.

Like the preceding FOR / NEXT command, the GOSUB / RETURN command uses a stack for its return address. If the subroutine is not allowed to complete normally: e.g., a GOTO lineno before a RETURN, the GOSUB address must be "popped" off the stack (see POP) or it could cause future errors.

To prevent accidental triggering of a subroutine (which normally follows the main program), place an END statement preceding the subroutine. The following program demonstrates the use of subroutines.

```
10 PRINT CHR$(125) : REM this clears the screen
20 REM EXAMPLE USE OF GOSUB / RETURN
30 X=100
40 GOSUB 1000
50 X=120
60 GOSUB 1000
70 X=50
80 GOSUB 1000
90 END
1000 Y=3*X
1010 X=X+Y
1020 PRINT X,Y
1030 RETURN
```

In the above program, the subroutine, beginning at line 1000, is called three times to compute and print out different values of X and Y. Below are the results of executing this program.

```
400      300
480      360
200      150
```

4.5 GOTO (G.)

       Format: [lineno] GOTO aexp

       Example: 100 GOTO 50
                     500 GOTO (X + Y)

The GOTO command is an unconditional branch statement just like the GOSUB command. They both immediately transfer program control to a target line number or arbitrary expression. However, you cannot RETURN from a GOTO, as you can with a GOSUB. If the target line number is non-existent, an error results. Any GOTO statement that branches to a preceding line may result in an "endless" loop. Statements following a GOTO statement will not be executed. Note that a conditional branching statement (see IF / THEN) can be used to break out of a GOTO loop. The following program illustrates uses the GOTO statement twice.

```
10 FINISH=70
20 PRINT : PRINT "ONE"
30 PRINT "TWO"
40 PRINT "THREE"
50 PRINT "FOUR"
60 PRINT "FIVE"
65 GOTO 100
70 PRINT "$$$$$$$$$$$$$$$$$$$"
80 PRINT "/./././././././././././."
90 PRINT "?????????????????"
95 END
100 PRINT "SIX"
110 PRINT "SEVEN"
120 PRINT "EIGHT"
130 PRINT "NINE"
140 PRINT "TEN"
150 GOTO FINISH
```

Upon execution, the numbers in the above listing will be listed first followed by the three rows of symbols. The symbols listed on lines 70, 80, and 90 are ignored temporarily while the program executes the GOTO 100 command. It proceeds with the printing of the numbers "SIX" through "TEN", then executes the second GOTO statement which transfers program control back to line 70. (This is just an example. This program could be rewritten so that no GOTO statements were used.) The program, when executed, looks like the following:

       ONE
       TWO
       THREE
       FOUR
       FIVE
       SIX

```
SEVEN
EIGHT
NINE
TEN
$$$$$$$$$$$$$$$$$$
/./././././././././././././.
??????????????????
```

NOTE: using anything other than a constant will make renumbering the
lineno in the GOTO (using RENUM) difficult. However, readability may be
markedly improved.

4.6 IF / THEN

      Format: IF aexp THEN lineno
                IF aexp THEN statement [:statement…]

      Example: IF X = 100 THEN 150
                IF A$ = "ATARI" THEN 200
                IF AA = 145 AND BB = 1 THEN PRINT
                      AA,BB
                IF X = 100 THEN X = 0

See also IF…ELSE…ENDIF discussion in the following section.

The IF / THEN statement is a conditional branch statement. This type of branch occurs only if certain conditions are met. These conditions may be either arithmetical or logical. If the aexp following the IF the THEN part of the statement. If, however, the aexp is false and/or zero, the rest of the statement is ignored and program control passes to the next numbered line.

In the format, IF aexp THEN lineno

lineno must be a constant (not an expression) specifying the line number to go to if the expression is true. If several statements occur after the THEN, separated by colons, then they will be executed if and only if the expression is true. Several IF statements may be nested on the same line. For example:

        100 IF X=5 THEN IF Y=3 THEN R=9: GOTO 200

The statements R=9 : GOTO 200 will be executed only if X=5 and Y=3. The statement Y=3 will be executed if X=5. The following program demonstrates the IF / THEN statement:

        100 GRAPHICS 0 : PRINT
        110 PRINT ,,"IF DEMO"
        120 PRINT : PRINT "ENTER A"; : INPUT A
        130 IF A=1 THEN 150 : REM Multiple Statements here will never be executed!!!
        140 PRINT : PRINT "A IS NOT 1, EXECUTION CONTINUES HERE WHEN EXPRESSION IS FALSE."
        150 IF A=1 THEN PRINT : PRINT "A=1?" : PRINT "YES, IT IS REALLY 1." : REM Multiple statements here will be executed only if A=1!!!
        160 PRINT : PRINT "EXECUTION CONTINUES HERE IF A <> 1 OR AFTER 'YES, IT IS REALLY 1' IS DISPLAYED."
        170 GOTO 100

Output of the above program is:

        IF DEMO

ENTER A   ?   (entered 2)
A   IS   NOT   1.   EXECUTION   CONTINUES   HERE   WHEN   THE
EXPRESSION IS FALSE.
EXECUTION   CONTINUES   HERE   IF   A<>1   OR   AFTER   'YES',   IT   IS
REALLY 1' IS DISPLAYED.

ENTER A   ?   (entered 1)
A=1
YES, IT IS REALLY 1.
EXECUTION CONTINUES HERE IF A <> 1 OR AFTER
'YES, IT IS REALLY 1' IS DISPLAYED.

4.7 IF…ELSE…ENDIF

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

      Example: 200 IF A>100:PRINT "TOO BIG"
                     210 A=100
                     220 ELSE:PRINT "A-OK"
                     230 ENDIF

                     1000 IF A>C : B=A : ELSE : B=C : ENDIF

BASIC XL makes available an exceptionally powerful conditional capability via IF…ELSE…ENDIF.

In the format given, if the expression is TRUE (evaluates as non-zero) then all statements between the following colon and the corresponding ELSE (if it exists) or ENDIF (if no ELSE exists) are executed; if ELSE exists, the statements between it and ENDIF are skipped.

If the expression is FALSE (evaluates to zero), then the statements (if any) between the colon and ELSE are skipped and those between ELSE and ENDIF are executed. If no ELSE exists, all statements through the ENDIF are skipped.

CAUTION: The colon following the aexp IS REQUIRED and MUST be followed by a statement. The word THEN is NOT ALLOWED in this format.

There may be any number (including zero) of statements and lines between the colon and the ELSE and between the ELSE and the ENDIF.

The second example above sets B to the larger of the values of A and C.

This IF structure may also be nested, as follows:

      100 IF A>B : REM SO FAR A IS BIGGER
      110   IF A>C : PRINT "A BIGGEST"
      120   ELSE : PRINT "C BIGGEST"
      130   ENDIF
      140 ELSE
      150   IF B>C : PRINT "B BIGGEST"
      160   ELSE : PRINT "C BIGGEST"
      170   ENDIF
      180 ENDIF

4.8 LET

Format: [LET] <assignment statement>

Example: LET GOTO=3.5
          LET LETTERS="a"
          LET AND$="*",A$,A$,A$,A$,A$

LET is an optional keyword which allows you to assign a value to a variable name which starts with or is identical to a reserved name. For example:

        10 LET GOSUBBER = 5
        20 LET PRINT = 7
        30 LET LET = PRINT + GOSUBBER
        40 PRINT PRINT,LET,GOSUBBER

will print out:

                7   12   5

There are a few keywords which CANNOT be used as variable names through the use of LET, including any function name and the NOT unary operator.

Here is an example of what will happen if you try to use one of the above as a variable name:

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

will print out: 1

If you LIST the program out you will see why. It lists "30 PRINT NOTE" as

        30 PRINT NOT E

because the interpreter does not allow NOT to start a variable name.

4.9 MOVE

        Format: MOVE aexp1,aexp2,aexp3

        Example: MOVE $D000, $8000, $400

CAUTION: be careful with this command!!

MOVE is a general purpose byte move utility which 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 the third aexp (the length) determines the order in which the bytes are moved, as follows:

        If the length is positive:
                (from) -> (to)
                (from+1) -> (to+1)
                …   …
                (from+len-1) -> (to+len-1)

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

        If the length is negative:
                (from+len-1) -> (to+len-1)
                (from+len-2) -> (to+len-2)
                …   …
                (from+1) -> (to+1)
                (from) -> (to)

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

4.10 ON…

        Format: ON aexp | GOTO | lineno [,lineno…]
                               | GOSUB |

        Example: 100 ON X GOTO 200,300,400
                    100 ON A GOSUB 1000,2000
                    100 ON SQR(X) GOTO 30,10,100

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

These two statements are also conditional branch statements like the IF / THEN statement. However, these two are more powerful. The aexp must evaluate to a positive number which is then rounded to the nearest positive integer (whole number) value up to 255. If the resulting number is 1, then program control passes to the first lineno in the list following the GOSUB or GOTO. If the resulting number is 2, program control passes to the second lineno in the list, and so on.

If the resulting number is 0 or is greater than the number of linenos in the list, the conditions are not met and program control passes to the next statement which may or may not be located on the same line. With ON / GOSUB, the selected subroutine is executed and then program control passes to the statement following the ON / GOSUB.

The following routine demonstrates the ON / GOTO statement:

```
10 X=X+1
20 ON X GOTO 100,200,300,400,500
30 IF X>5 THEN PRINT "COMPLETE.":END
40 GOTO 10
50 END
100 PRINT "NOW WORKING AT LINE 100":GOTO 10
200 PRINT "NOW WORKING AT LINE 200":GOTO 10
300 PRINT "NOW WORKING AT LINE 300":GOTO 10
400 PRINT "NOW WORKING AT LINE 400":GOTO 10
500 PRINT "NOW WORKING AT LINE 500":GOTO 10
```

When the program is executed, it looks like the following:

NOW WORKING AT LINE 100
NOW WORKING AT LINE 200
NOW WORKING AT LINE 300
NOW WORKING AT LINE 400
NOW WORKING AT LINE 500
COMPLETE.

       Format: POP

       Example: 1000 POP

In the description of the FOR / NEXT statement, the stack was defined as a group of memory addresses reserved for return addresses. The top entry in the stack controls the number of loops to be executed and the RETURN target line for a GOSUB. If a subroutine is not terminated by a RETURN statement, the top memory location of the stack is still loaded with some numbers. If another GOSUB is executed, that top location needs to be cleared. To prepare the stack for a new GOSUB, use a POP to clear the data from the top location in the stack.

The POP command could be used in the following ways:

1)   In a FOR or WHILE statement, when you wish jump out of the loop before it has executed its specified number of times (e.g., if you are searching through a lot of data for a specific item, you can leave the loop early by POPping the stack, and then using GOTO to continue execution after the NEXT). Example:

```
10 FLAG = 1
20 WHILE FLAG
30    INPUT FLAG
40    IF FLAG < 0 THEN POP : GOTO 70
50    PRINT "IN THE WHILE LOOP"
60    ENDWHILE
70 END
```

2)   After a subroutine (GOSUB) which does not give control back to the main program through the use of a RETURN. The following example illustrates this instance:

```
100 REM POP Demo
110 N = 1 : GOSUB 800
120 N = 2 : GOSUB 800
130 END
800 PRINT "At Line 800"
810 GOSUB 900
820 PRINT "At Line 820"
830 RETURN
900 PRINT "At Line 900"
910 IF N = 2 THEN POP
920 RETURN
```

4.12 RESTORE (RES.)

       Format: RESTORE [aexp]

       Example: 100 RESTORE
                    200 RESTORE X+2

BASIC XL contains an internal "pointer" that keeps track of the DATA statement item to be read next. When used without the optional aexp, the RESTORE statement resets that pointer to the first DATA item in the program. When used with the optional aexp, the RESTORE statement sets the pointer to the first DATA item on the line specified by the value of the aexp.

This statement permits repetitive use of the same data, as shown in the following example:

       10 FOR N=2 TO 1 STEP -1
       20   RESTORE 80+N
       30   READ A, B
       40   M=A+B
       50   PRINT "TOTAL EQUALS ";M
       60 NEXT N
       70 END
       81 DATA 30,15
       82 DATA 10,20

On the first pass through the loop, A will be 10 and B will be 20 so the total in line 50 will print: TOTAL EQUALS 30, but on the second pass, A will equal 30 and B will equal 15, so the PRINT statement in line 50 will display: TOTAL EQUALS 45.

4.12 TRAP (T.)

Format: TRAP aexp

Example: 100 TRAP 120

The TRAP statement is used to direct the program to a specified line number if an error is detected. Without a TRAP statement, 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 TRAP statement should be placed at the beginning of the section of code that handles input from the keyboard so that the TRAP is reset after each error.

You can find out the error number using the ERR function with an argument of 0, and fine out the lineno on which the error occurred by using the ERR function with an argument of 1 (see section 6.6.4 for a more detailed discussion of ERR).

Alternatively, PEEK(195) will give you the error number, and DPEEK(186) will give you the number of the line where the error occurred.

A TRAP may be disabled by executing a TRAP statement with an aexp whose value is zero (0), or between 32768 and 65535 (e.g., TRAP 40000).

4.14 WHILE…ENDWHILE

     Format: WHILE aexp : <statement> : ENDWHILE

     Example: 100 A=3
                110 WHILE A : PRINT A
                120   A=A-1 : ENDWHILE

With WHILE, the BASIC XL user has yet another powerful control structure available. So long as the aexp of WHILE remains non-zero, all statements between WHILE and ENDWHILE are executed.

Example: WHILE 1 : ….
                 The loop executes forever

Example: WHILE 0 : ….
                 The loop will never execute

CAUTION: Do not GOTO out of a WHILE loop or a nesting error will likely result (unless you use POP first).

NOTE: The aexp is only tested at the top of each passage through the loop.

This chapter describes the input / output devices and how data is moved between them. The commands explained in this chapter are those that allow access to the input / output devices. The input commands are those associated with putting data into RAM and the devices geared for accepting input. The output commands are those associated with retrieving data from RAM and the devices geared for generating output.

The commands described in this chapter are:

BGET	DIR	LPRINT	PROTECT	SAVE
BPUT	ENTER	NOTE	PUT	STATUS
CLOAD	ERASE	OPEN	READ	TAB
CLOSE	GET	POINT	RENAME	UNPROTECT
CSAVE	INPUT	PRINT	RGET	XIO
DATA	LOAD	PRINT USING	RPUT	

5.1 Comments and Notations

The Atari Personal Computer considers everything except the guts of the computer (i.e. the RAM, ROM, and processing chips) to be external devices. Some of these devices come with the computer, for example the Keyboard and the 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 name used as 'filespec' in the BASIC XL commands:

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) – handles 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 specify a file name along with the device name (see you 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 XL program. When you specify no channel while doing I / O, E: is used because the channel defaults to 0, which is the channel BASIC XL 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 four RS-232 Serial Ports on the Atari 850 Interface – handle both Input and Output. These devices enable the Atari system to interface to RS-232 compatible serial devices like terminals, plotters, and modems.

S: The Screen Display (either TV or Monitor) – handles both Input and Output. This device allows you to do I / O of either characters or graphics points with 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 the way in which they work is different, each device has to tell the computer how it operates. This is 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 in the Central Input Output Processor (CIO). It's 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 (section 5.16) the device on one of the CIO's eight channels (numbered 0-7). When you then want to do I / O involving the OPENed device, you 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 either be a literal string (i.e., enclosed in quotes), a string of characters (not in quotes), or a string variable.

IF IOCS #7 is in use, it will prevent LPRINT or some of the other BASIC I / O statements from being performed.

In the examples in the following sections, you will often see wildcard characters * and 7 in the filespec. For information on the use of these, see your DOS manual.

5.2 BGET

       Format: BGET #channel, aexp1, aexp2

       Example: (see below)

BGET gets "aexp2" bytes from the device or file specified by "channel" and stores them at address "aexp1".

NOTE: The address may be a memory address. For example, a screen full of data could be displayed in this manner. Or the address may be the address of a string. In this case BGET does not change the length of the string: this is a user's responsibility.

Example: 10 DIM A$(1025)
                20 BGET #5, ADR(A$),1024
                30 A$(1025) = CHR$(0)

This program segment will get 1024 bytes from the file or device associated with file number 5 and store it in A$. Statement 30 sets the length of A$ to 1025.

NOTE: No error checking is done on the address or length so care must be taken when using this statement.

For another example using BGET, see section 5.31.

5.3 BPUT

Format:　BPUT #channel, aexp1, aexp2

Example:　BPUT #5, ADR(A$), LEN(A$)

BPUT outputs a block of data to the device or file specified by "channel". The block of data starts at address "aexp1" for a length of "aexp2".

NOTE:　The address may be a memory address.　For example, the whole screen might be saved.　Or the address may be the address of a string obtained using the ADR function.

The example above writes the block of data contained in the string A$ to the file or device associated with channel number 5.

NOTE:　Nothing is written to the file which indicates the length of the data written.　You are advised to write fixed-length data to make the rereading process simpler.

5.4 CLOAD

Format:　CLOAD

Example:　CLOAD
　　　　　100 CLOAD

This command can be used in either Direct or Deferred mode to load a program from cassette tape into RAM for execution.　On entering CLOAD, one bell rings 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 Program Recorder Manual.

5.5 CLOSE (CL.)

Format:　CLOSE #channel

Example:　CLOSE #4
　　　　　100 CLOSE #1

The CLOSE command is used to close a CIO channel which has been previously OPENed to allow I / O with 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.

NOTE:　END will also close all channels (i.e., files).

5.6 CSAVE (CS.)

    Format: CSAVE

    Example: CSAVE
         100 CSAVE
         100 CS.

This command is usually used in Direct mode to save a RAM-resident program onto cassette tape. CSAVE saves the tokenized version of the program. On entering CSAVE two bells ring to indicate that the PLAY and RECORD buttons must be pressed followed by <RETURN>. RECORD buttons must be pressed followed by <RETURN>. Do not, however, press these buttons until the tape has been positioned. It is faster to save a program using this command rather than a SAVE "C" (See SAVE) because short inter-record gaps are used.

NOTE: Tapes saved using the two commands SAVE and CSAVE are not compatible.

NOTE: Due to a flaw in the Atari OS ROMs, it may be necessary on some machines to enter an LPRINT (See LPRINT) before using CSAVE. Otherwise, CSAVE may not work properly.

For specific instructions on how to connect and operate the hardware, cue the tape, etc., see the Atari 410 Program Recorder Manual.

5.7 DATA (D.)

    Format: DATA adata [,adata]

    Example: 100 DATA 12,13,14,15,16
         200 DATA GEORGE, EVELYN, MIKE, BECKY
         300 DATA "DATA with a comma, in quotes"

The DATA command is used in conjunction with the READ command (see section 5.22) to access elements in a data list. A DATA command may be anywhere in a program, but it must contain as many pieces of data as there are defined in the READ command; otherwise an "out of data" error is displayed on the screen.

NOTE: All characters except command and <RETURN> are allowed. However, if you put the data in quotes, then all characters except double quote and <RETURN> are legal.

5.8 DIR

>   Format: DIR [filespec]

>   Example: DIR D:*.COM
>   DIR FILES
>   DIR "D2:TEST*.B*"

The DIR Command is used to list the contents of a disk directory to the screen. It is very similar to the OS/A+ and DOS XL 'DIR' command. If no filespec is given, all files on D1: are displayed.

The first example will display all files on D1: which end with .COM.

The second example shows a string variable being used as a 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 disk drive 2 which match TEST*.B*.

NOTE: Dir must be used as the last (or only) command on a line.

5.9 ENTER (E.)

>   Format: ENTER filespec

>   Example: ENTER "C:"
>   ENTER D2:DEMOPR.INS
>   ENTER FILE$

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 SAVEd or CSAVEd. To use this command, you simply need to give the filespec of the program.

NOTE: Whereas both LOAD and CLOAD clear the old program from memory before reading in the new one, ENTER does not, and so is useful when trying to merge programs together.

ENTER can be modified using the SET command. For an example of this, see section 3.15, example 3.

5.10 ERASE

      Format: ERASE filespec

      Example: 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 .BAK (back-up) files on disk drive 1. The second example would erase all files matching TEST?.SAV on disk drive 2. This command is similar to the OS/A+ and DOS XL ERAse, but there are no default file specifiers.

5.11 GET

      Format: GET #channel,avar

      Example: 100 GET #0,X

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

For a program example using GET, see section 5.31.

5.12 INPUT (I.)

      Format: INPUT [#chan,] |avar [avar[,(avar)…]|
                                    | svar |

      Example: 100 INPUT X
                   100 INPUT N$
                   100 INPUT X,Y,Z(4)
                   100 INPUT ARRSTR$(5;)
                   100 PRINT "ENTER A VALUE OF X"
                   110 PRINT X

INPUT is used to read in various data. With it you can input either one or more numbers, or a string. If you are inputting a group of numbers, the first number will go into the first avar specified, the second number into the second avar, and so on.

NOTE: In BASIC XL the avar may be an array element, and the svar may be a string array element.

If a channel number is specified (followed by a comma), then no "?" prompt is given. This allows you to create your own prompts, as shown in the following example:

      100 PRINT "command>>   ";
      110 INPUT #0, COMMAND$

The statement 'INPUT #0, COMMAND$' inputs a string from channel 0 (E:), without printing out a '?' first.

NOTE: If the user's sole response to an INPUT prompt is <CTRL>C <RETURN>, a special error (number 27) will be issued by INPUT. This can be useful in data entry manipulations.

If an INPUT request is made for more than one numeric variable, the user may respond with several values separated by commas or may type in single number on each line, followed by <RETURN>.

In the latter case, BASIC XL will prompt with a double question mark to indicate that more input is needed. When a string is requested, it must be typed on a line by itself (or, if combined with numeric input, as the last item on the line).

OSS strongly recommends 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).

5.12.1 Advanced use of INPUT

Format: INPUT "string", var [,var…]

Example: 100 INPUT "3 VALUES>> ",V(1),V(2),V(3)

BASIC XL allows you to include a prompt with the INPUT command to produce easier to use programs, without having to use the ";" option mentioned in the previous section. The string given in the above format ALWAYS replaces the default "?" prompt.

NOTE: No channel number may be used when the literal prompt is present.

NOTE: In the example above, if the user typed in only a single value followed by a <RETURN>, he would be reprompted by BASIC XL with a "??", but see chapter 3 for variations available via SET.

5.13 LOAD (LO.)

Format: Load filespec

Example: LOAD D1:GAME1.BXL
          100 LOAD "C:"

LOAD allows you to load the SAVEd version of a program into memory from any device. It will not work properly with programs saved using LIST or CSAVE, so they have their own loading command (see ENTER and CLOAD).

5.14 LPRINT (LP.)

Format: LPRINT [exp][| ; | exp…]

Example: LPRINT "PROGRAM TO CALCULATE X"

This statement causes the computer to print data on the line printer rather than on the screen. It can be used in either Direct or Deferred mode, and requires no device specified, no OPEN, or no CLOSE statement.

NOTE: The semicolon and comma options are discussed in section 5.18, PRINT.

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

5.15 NOTE (NO.)

      Format: NOTE #chan, avar,avar

      Example: 100 NOTE #1,X,Y

This command is used to store the current disk sector number in the first avar and the current byte number within the sector in the second avar. This is the current read or write position in the specified file where the next byte to be read or written is located.

5.16 OPEN (O.)

      Format: OPEN #chan,aexp1,aexp2,filespec

      Example: 100 OPEN #2,8,0,"C:"
                     100 A$ = "D1:TEST.DAT"
                     110 OPEN #2,8,0,A$

As mentioned in section 5.1, 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 number codes are described in the following table:
	
	   aexp1 Meaning
	     4 Input only
	     6 Read disk directory only
	     8 Output only
	     9 Output append. This mode allows you to append to already existing disk files.
	   12 Input and Output

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.

5.17 POINT (P.)

       Format: POINT #chan,avar,avar

       Example: 100 POINT #2,A,B

This command is used when reading a file into RAM. The first avar specifies the sector number and the second avar specifies the byte within that sector where the next byte will be read or written. 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.

5.18 PRINT (PR OR ?)

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

       Example: PRINT
                PRINT X,Y,Z, A$
                100 PRINT "THE VALUE OF X IS ";X
                100 PRINT "COMMAS", "CAUSE", "COLUMNS"
                100 PRINT #3,A$
                100 PRINT #0;"$";HEX(X);" IS ";X

The PRINT command is used in either Direct or Deferred mode to output data. In Direct mode, this command prints whatever information is contained between the quotation marks exactly as it appears. In the second example, PRINT X,Y,Z,A$, the screen will display the current values of X,Y,Z and A$ as they appear in the RAM-resident program. 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. A semicolon causes the next aexp or sexp to be placed immediately after the preceding expression with no spacing. Therefore, in the third example a space is placed before the ending quotation mark so the value of X will not be placed immediately after the word "IS".

If no command 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.

5.19 PRINT USING

Format: PRINT [#ch;]USING sexp,exp [,exp...]

Example: (see below)

PRINT USING allows the user to specify a format for the output to the device of file associated with "ch" (or to the screen). The format string "sexp" contains one or more format fields. Each format field tells how an expression from the expression list is to be printed. Valid format field characters are:

   # & * + - $, . % ! /

Non-format characters terminate a format field and are printed as they appear.

Example:

   1) 100 PRINT USING "## ###X#",12,315,7

   2) 100 DIM A$(10) : A$="## ###X#"
       200 PRINT USING A$,12,315,7

   Both 1) and 2) will print

   12 315X7

Where a blank separates the first two numbers and an X separates the last two.

Numeric Formats:

The format characters for numeric format fields are:

   # & * + - $, .

DIGITS (# & *)

Digits are represented by:

   # & *

- Indicates fill with leading blanks
& - Indicates fill with leading zeroes
* - Indicates fill with leading asterisks

If the number of digits in the expression is less than the number of digits specified in the format then the digits are right justified in the field and preceded with the proper fill character.

NOTE: In all the following examples b is used to represent a blank.

Example:

Value	Format Field	Print Out
1	###	bb1
12	###	b12
123	###	123
1234	###	234
12	&&&	012
12	***	*12

DECIMAL POINT (.)

A decimal point in the format field indicates that a decimal point be printed at that location in the number. All digit positions that follow the decimal point are filled with digits. If the expression contains fewer fractional digits than are indicated in the format, then zeroes are printed in the extra positions. If the expression contains more fractional digits than indicated in the format, then the expression is rounded so that the number of fractional digits is equal to the number of format positions specified.

A second decimal point is treated as a non-format character.

Example:

Value	Format Field	Print Out
123.456	###.##	123.46
4.7	###.##	bb4.70
12.35	##.##.	12.35.

COMMA (,)

A comma in the format field indicates that a comma be printed at that location in the number. If the format specifies a comma be printed at a position that is preceded only by fill characters (0 b *) then the appropriate fill character will be printed instead of the comma.

The comma is a valid format character only to the left of the decimal point. When a comma appears to the right of a decimal point, it becomes a non-format character. It terminates the format field and is printed like a non-format character.

Example:

Value	Format Field	Print Out
5216	##,###	b5.216
3	##,###	bbbbb3
4175	**,***	*4,175
3	&&,&&&	000003
42.71	##.##,	42.71,

SIGNS (+ -)

A plus sign in a format field indicates that the sign of the number is to be printed. A minus sign indicates that a minus sign is to be printed if the number is negative and a blank if the number is positive.

Signs may be either fixed, floating or trailing.

A fixed sign must appear as the first character of a format field.

Example:

Value	Format Field	Print Out
43.7	+###.#	+b43.7
-43.7	+***.*	-b43.7
23.58	-&&&.&&	b023.58
-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 one digit.

Example:

Value	Format Field	Print Out
3.75	++++.$$	bb+3.75
3.75	----.##	bbb3.75
-3.75	----.##	bb-3.75

A trailing sign can appear only after a decimal point. It terminates the format and prints the appropriate sign (or blank).

Example:

Value	Format Field	Print Out
43.17	***.**+	*43.17+
43.17	&&&.&&-	043.17b
-43.17	###.##+	b43.17-

DOLLAR SIGN ($)

A dollar sign can be either fixed or floating, and indicates that a $ is to be printed.

A fixed dollar sign must be either the first or second character in the format field. If it is the second character then + or – must be the first.

Example:

Value	Format Field	Print Out
34.2	$##.##	$34.20
34.2	+$##.##	+$34.20
-34.2	+$###.##	-$34.20

Floating dollar signs must start as either the first or second character in the format field and continue to the decimal point. If the floating dollar signs start as the second character then + or – must be the first. Each dollar sign after the first also represents one digit.

Example:

Value	Format Field	Print Out
34.2	$$$$$.##	bb$34.20
34.2	+$$$$$.##	+bb$34.20
1572563.41	$$,$$$,$$$.##+	$1,572,563.41+

NOTE: There can only be one floating character per format field.

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

<u>String Formats</u>

The format characters for string format fields are:

> % - 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, than the string is truncated.

Example:

Value	Format Field	Print Out
ABC	%%%%	bABC
ABC	!!!!	ABCb
ABC	%%	AB
ABC	!!	AB

ESCAPE CHARACTER (/)

The escape character (/) does not terminate the format field but will cause the next character to be printed, thus allowing the user to insert a character in the middle of the printing of a number.

Example: PRINT USING "###/-###",2551472
        prints   255-1472

Example: 100 AREA = 408
            200 NUM = 2551472
            300 PHONE = (AREA*1E+7)+NUM
            400 DIM F$(20)
            500 F$ = "(###/)###/-####"
            600 PRINT USING F$, PHONE
            700 END

the result: (408)255-1472

NOTE: Improperly specified format fields can give some very strange results.

NOTE: The function of "," and ";" in PRINT are overridden in the expression list of PRINT USING, but when file number "ch" is given then the following "," or ";" have the same meaning as in PRINT. So to avoid an initial tabbing, use a semicolon (;).

Example: PRINT USING "## /* #=###",12,5,5*12

            12 * 5 = 60

Example: PRINT USING "TOTAL=##.#+",72,68

            TOTAL=72.7+

Example: 100 DIM A$(10) : A$="TOTAL="
            200 DIM F$(10) : F$="!!!!!!##.#+"
            300 PRINT USING F$,A$,72.68

            TOTAL=72.7+

NOTE: If there are more expressions in the expression list than there are format fields, the format fields will be reused.

Example: PRINT USING "XX##",25,19,7

            will print   XX25XX19XXb7

WARNING: A format string must contain at least one format field. If the format string contains only non-format characters, those characters will be printed repeatedly in the search for a format field.

5.20 PROTECT

       Format: PROTECT filespec

       Example: PROTECT D:*.COM
                   100 PROTECT "D2:JUNK.BXL"

The PROTECT allows you to protect your programs stored on disk from being erased or overwritten. This command is very similar to the OS/A+ and DOS XL PROtect command, except that there are no default file specifications.

5.21 PUT (PU.)

Put is the opposite of GET in that it outputs a single byte of information whereas GET inputs a single byte of information. The data output is aexp, and it is put to the device specified by chan.

NOTE: For a program example using PUT, see section 5.31

5.22 READ

        Format: READ var [,var…]

        Example: 100 READ A,B,C,D,E
                   110 DATA 12,13,14,15,16

                   100 READ A$,B$,C$,D$,E$
                   110 DATA EMBEE, EVELYN, CARLA

The READ command is always used in conjunction with the DATA command. Its function is simply to read the next piece of data out of the DATA list and put it into one of the variables specified. If a group of variables are used, then the first piece of available data (see RESTORE, 4.12) is put into the first variable given, the second piece of data into the second variable given, and so on.

The type of the variable in the READ statement (svar or avar) must correspond to the type of the data which being read.

If the second example above was executed as a program with no additional lines, an error would result since there are fewer data items than variables to be READ.

The following program totals a list of numbers in a DATA statement:

        10 FOR N=1 TO 5
        20 READ D
        30 M=M+D
        40 NEXT N
        50 PRINT "SUM TOTAL EQUALS ":M
        60 END
        70 DATA 30,15,106, 87,17

The program, when executed, will print the statement:

        SUM TOTAL EQUALS 255.

NOTE: A Direct mode READ will only read data if a DATA statement exists in the program or on the line following the READ.

5.23 RENAME

        Format: RENAME "filespec,filename"

        Example: RENAME "DS:NEW.DAT, OLD.BAK"

RENAME allows you to rename file(s) from BASIC XL. Note that the comma shown MUST be imbedded in the string used as the file parameter.

CAUTION: It is strongly suggested that wild cards (* and ?) NOT be used when RENAMEing. Also, the second filename may NOT include the disk specifier (Dn:).

5.24 RGET

        Format: RGET #ch, | svar [,svar…] |
                                      | avar [,avar…] |

        Example: (see below)

RGET allows the user to retrieve fixed length records from the device or file associated with file number "ch" and assign the values to string or numeric variables.

NOTE: The type of the element in the file must match the type of the variable (i.e. They must both be strings or both be numeric).

Example: 1) 100 RPUT #3,C
                  :
         2) 200 RGET #1,A$

If 1) is a statement in a program used to generate a file and 2) is a statement in another program use to read the same file, an error will result, since 'C' is a numeric variable and 'A$' is a string variable.

NOTE: When the type of element is string, then the DIMensioned length of the element in the file must be equal to the DIMensioned length of the string variable.

Example: 1) 100 DIM A$(100)
                  :
              800 RPUT #3,A$

         2) 100 DIM X#(200)
                  :
              800 RGET #2,X$

If 1) is a section of a program used to write a file and 2) is a section of another program used to read the same file, then an error will occur as a result of the difference in DIM values.

NOTE: RGET sets the correct length for a string variable (the length of a string variable becomes the actual length of the string that was RPUT – not necessarily the DIM length).

Example: 1) 100 DIM A$(10)
    200 A$ = "ABCDE"
     :
    800 RPUT #4,A$

    2) 100 DIM X$(10)
     200 X$ = "HI"
      :
     800 RGET #6,X$
     900 PRINT LEN(X$),X$

If 1) is a section of a program used to create a file and 2) is a section of another program used to read the file then it will print:

       5   ABCDE

5.25 RPUT

    Format: RPUT #ch, exp [,exp…]

    Example: (see below)

RPUT allows the user to output fixed length records to the device or file associated with "ch". Each "exp" creates an element in the record.

NOTE: A numeric element consists of one byte which indicates a numeric type element and 6 bytes of numeric data in floating point format.

A string element consists of one byte which indicates a string type element 2 bytes of string length, 2 bytes of DIMensioned length, and then X bytes where X is the DIMensioned length of the string.

Example: 100 DIM A$(6)
       200 A$ = "XY"
       300 RPUT #3,B,A$,10

puts 3 elements to the device or file associated with file number 3. The first element is numeric (the value of B). The second element is a string (A$) and the third is a numeric (10). The record will be 26 bytes long. (7 bytes for each numeric, 5 bytes for the string header and 6 bytes (the DIM length) of string data).

5.26 SAVE (S.)

      Format: SAVE filespec

      Example: SAVE D1:YVONNE.PAT
                100 SAVE "C:"

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

5.27 STATUS (ST.)

      Format: STATUS #chan,avar

      Example: 350 STATUS #1,Z

The STATUS command calls the STATUS routine for the specified device (chan). The status of the STATUS command (see ERROR MESSAGES, Appendix B) is stored in the specified variable (avar). This may be useful for devices such as the RS-232 interface.

5.28 TAB

      Format: TAB [#ch,] aexp

      Example: TAB #2,20

TAB outputs spaces to the device or file specified by ch (or the screen) up to column number "aexp". The first column is column 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 to AUX2 of the IOCB. (See OS documentation).

NOTE: If "aexp" is less than the current column count, a carriage return is output and then spaces are put out up to column "aexp".

5.29 UNPROTECT (UNP.)

      Format: UNPROTECT filespec

      Example: 100 UNPROTECT "D2:JUNK.BAS"
                UNP. D:JUNK

The UNPROTECT command allows you to unprotect disk files which have been protected using the PROTECT command. This command is very similar to the OS/A+ and DOS XL command UNProtect, but there are no default file specifications in the BASIC XL version.

5.30 XIO (X.)

        Format: XIO cmdno, #chan,aexp1,aexp2,"filespec"

        Example: XIO 10,#6,0,0,"S:"

The XIO command is a general input/output statement used for special operations. The parameters for this command are defined as follows:

cmdno   Number for stands for the particular command to be performed.

cmdno	operation	example
3	OPEN	Same as BASIC OPEN
5	GET RECORD	| These 4 commands are
7	GET CHARACTERS	| similar to BASIC INPUT,
9	PUT RECORD	| GET, PRINT, and PUT,
11	PUT CHARACTERS	| respectively.
12	CLOSE	Same as BASIC CLOSE
13	STATUS REQUEST	Same as BASIC STATUS
17	DRAW LINE	Same as BASIC DRAWTO
18	FILL	See Section 9
32	RENAME	XIO 32,#1,0,0,"D:TEMP,CAROL"
33	DELETE	XIO 33,#1,0,0,"D:TEMP.BAS"
35	LOCK FILE	XIO 35,#1,0,0,"D:TEMP.BAS"
36	UNLOCK FILE	XIO 36,#1,0,0,"D:TEMP.BAS"
37	POINT	Same as BASIC POINT
38	NOTE	Same as BASIC NOTE
254	FORMAT	XIO 254,#1,0,0,"D2:"

chan   Device number (same as in OPEN). Most of the time it is ignored, but must be preceded by #.

aexp1   Two auxiliary control bytes. Their usage depends on the
aexp2   particular device and command. In most cases, they are unused and are set to 0.

filespec String expression that specified the device. Must be enclosed in quotation marks. Although some commands do not look at the filespec, it must still be included in the statement.

NOTE: It is highly recommended that the BASIC XL user avoid XIO cmdno's 3,5,7,9,11,12,17,37 and 38. BASIC XL users should find all these, as well as cmdno's 32 thru 36, totally unnecessary.

5.31 An Example Program

The following subroutine reads in a binary file using OPEN, GET, BGET, CLOSE, and PRINT.

NOTE: Lines 1020 through 1030 test the file to see if it is segmented, so you can load in multi-segment files with the subroutine.

```
1000 TRAP 1090
1010 OPEN #1,4,0,"D:FILE.OBJ"
1020 GET #1,L : GET #1,H
1030 IF L=$FF AND H=$FF THEN GET #1,L : GET #1,H
1040 START = H*256+L
1050 GET #1,L : GET #1,H
1060 FINISH = H*256+L
1070 BGET #1, START, FINISH – START + 1
1080 GOTO 1020
1090 IF ERR(0)=136 THEN CLOSE #1 : RETURN
1100 PRINT "UNEXPECTED ERROR #";ERR(0);" AT LINE "; ERR(1)
1110 STOP
```

A function performs a computation and returns the result (usually a number) for either a print-out or additional computational use. Each function described in this chapter may be used in either Direct or Deferred mode.

This chapter describes the following functions:

Arithmetic Functions			Trigonometric Functions	
ABS	INT	RND	ATN	RAD
CLOG	LOG	SGN	COS	SIN
EXP	RANDOM	SQR	DEG	

String Functions			Game Controller Functions		
ASC	LEFT$	RIGHT$	HSTICK	PTRIG	VSTICK
CHR$	LEN	STR$	PADDLE	STICK	
FIND	MID$	VAL	PEN	STRIG	

Player/Missile Functions		Special Purpose Functions			
BUMP	PMADR	ADR	ERR	PEEK	TAB
		DPEEK	FRE	POKE	USR
		DPOKE	HEX$	SYS	

6.1 Arithmetic Functions

6.1.1 ABS

      Format: ABS(aexp)

      Example: 100 AB = ABS(-190)

Returns the absolute value of a number without regard to whether it is positive or negative. The returned value is always positive.

6.1.2 CLOG

      Format: CLOG (aexp)

      Example: 100 C = CLOG(83)

Returns the logarithm to the base 10 of the variable or expression in parentheses. CLOG(0) gives an error, and CLOG(1) is 0.

6.1.3 EXP

     Format: EXP(aexp)

     Example: 100 PRINT EXP(3)

Returns the value of e (approximately 2.71828283), raised to the power specified by the expression in parentheses. In the example given above, the number returned is 20.0855365.

6.1.4 INT

     Format: INT(aexp)

     Example: 100 I = INT(3.445) : REM I now = 3
                        100 X = INT(-14.66778) : REM X now = -15

Returns the greatest integer less than or equal to the value of the expression. This is true whether the expression evaluates to a positive or negative number. Thus, in our first example above, I is used to store the number 3. In the second example, X is used to store the number -15 (the first whole number that is less than or equal to -14.66778). This INT function should not be confused with the function used on calculators that simply truncates all decimal places.

6.1.5 LOG

     Format: LOG(aexp)

     Example: 100 L = LOG(67.89/2.57)

Returns the natural logarithm of the number or expression in parentheses. LOG (0) gives an error, and LOG(1) is 0.

6.1.6 RANDOM

     Format: RANDOM(aexp1[,aexp2])

     Example: 10 X = RANDOM(99)
                       10 Y = RANDOM(20,30)

The RANDOM function allows you to access to a random number generator which does more than return a number between 0 and 1, as RND does. When used with one aexp (as in the first example), the value returned will be between 0 and the aexp value, inclusive. When used with two aexps (as in the second example), the value returned will be between the value of the first aexp and the value of the second aexp, inclusive.

6.1.7 RND

      Format: RND(aexp)

      Example: 10 A = RND(0)

Returns a hardware-generated random number between 0 and 1, but never returns 1. The variable or expression in parentheses following RND is a dummy and has no effect on the numbers returned. However, the dummy expression must be included.

6.1.8 SGN

      Format: SGN(aexp)

      Example: 100 X = SGN(-199) : REM -1 is returned

Returns a -1 if aexp evaluates to a negative number: a 0 if aexp evaluates to 0, or a 1 if aexp evaluates to a positive number.

6.1.9 SQR

      Format: SQR(aexp)

      Example: 100 PRINT SQR(100) : REM 10 is printed

Returns the square root of the aexp which must be positive.

6.1.10 An Example Program

The following program prints out some information on an INPUTted number, using the arithmetic functions ABS, INT, SQR, CLOG, LOG, and EXP.

```
100 GRAPHICS 1 : REM set up screen
110 PRINT "Number to manipulate> ";
120 INPUT #0, X : REM get the number
130 PRINT #6; ASC$(125) : REM clear screen
140 PRINT #6; "ABS.: ";ABS(X) : REM absolute value
150 PRINT #6
160 PRINT #6; "INT.: "; INT(X) : REM integer value
170 PRINT #6
180 PRINT #6; "SQRT: "; SQR(ABS(X)) : REM square root
190 PRINT #6
200 PRINT #6; "CLOG: "; CLOG(ABS(X)) : REM common log
210 PRINT #6
220 PRINT #6; "NLOG: "; LOG(ABS(X)) : REM natural log (in)
230 PRINT #6
240 PRINT #6; "EXP: "; EXP(X) : REM exponential (e^X)
250 GOTO 110
```

6.2 Trigonometric Functions

6.2.1 ATN

      Format: ATN(aexp)

      Example: 100 X = ATN(1.0)

Returns the arctangent of the variable or expression in parentheses. If in DEG mode (see section 6.2.3), the returned value is given is degrees, otherwise it is given in radians.

6.2.2 COS

      Format: COS(aexp)

      Example: 100 C = COS(X+Y+Z)

Returns the trigonometric cosine of the expression in parentheses. The expression is evaluated as an angle in radian terms unless the DEG command has been used.

6.2.3 DEG and RAD

      Format: DEG
                  RAD

      Example: 100 DEG
                  100 RAD

These two statements allow the programmer to specify degrees or radians for trigonometric function computations. The computer defaults to radians unless DEG is specified. Once the DEG statement has been executed, RAD must be used to return to radians.

See Appendix E for the additional trigonometric functions that can be derived.

6.2.4 SIN

      Format: SIN(aexp)

      Example: 100 X = SIN(Y)

This function returns the trigonometric sine of aexp. The expression is evaluated as an angle in radian terms unless the DEG command has been used.

6.2.5 An Example Program

The following program demonstrates the use of DEG, COS, and SIN by plotting three concentric circles on the screen.

```
10 GRAPHICS 7 : REM set up screen
20 DEG : REM degree mode for trig functions
30   FOR J=1 TO 3 : REM 3 circles
40   COLOR J : REM each circle a different color
50     FOR I=1 TO 360 : REM plot each point in a full circle
60     PLOT 80+INT(J*10*COS( I )), 40+INT(J*10*SIN( I ))
70     NEXT I
80   NEXT J
```

6.3 String Functions

6.3.1 ASC

    Format: ASC(sexp)

    Example: 100 A = ASC(A$)

This function returns the ATASCII code number for the first character of the string expression (sexp). This function can be used in either Direct or Deferred mode.

If A$ = "ABC", then
        ASC(A$) produces 65
        ASC(A$(2)) produces 66

6.3.2 CHR$

    Format: CHR$(aexp)

    Example: 100 PRINT CHR$(65)
               100 A$ = CHR$(65)

This character string function returns the character, in string format, represented by the ATASCII code number in parentheses. 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 I=0 TO 25
        20 PRINT CHR$(ASC("A")+I);CHR$(ASC("a")+I
        30 NEXT I

NOTE: There can be only one STR$ and only one CHR$ in a logical comparison. (This is because BASIC XL uses a buffer in a fixed location to create the temporary string which both of these functions produce, and there is only one such buffer.

6.3.3 FIND

Format: FIND(sexp1,sexp2,aexp)

Example: PRINT FIND ("ABCDXXXXABC","BC","N")

FIND is an efficient, speedy way to determining whether any given substring is contained in any given master string.

FIND will search sexp1, starting at position aexp, for 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 N=1
        9 if N>2 and N<10
        0 if N>=10

More Examples:

1)   10 DIM A$(1)
        20 PRINT "INPUT A SINGLE LETTER:"
        30 PRINT "Change/Erase/List"
        40 INPUT "CHOICE ?",A$
        50 ON FIND("CEL",A$,0) GOTO 100,200,300

An easy way to have a vector from a menu choice:

2)   100 DIM A$(10): A$="ABCDEFGHIJ"
        110 PRINT FIND (A$,"E",3)
        120 PRINT FIND (A$(3),"E",0)

Line 110 will print "5" while 120 will print "3". Remember, the position returned is relative to the start of the specified string.

3)   100 INPUT "20 CHARACTERS, PLEASE:",A$
        110 ST=0
        120 F=FIND(A$,"A",ST):IF F=0 THEN STOP
        130 IF A$(F+1,F+1) <> "B" AND A$(F+1,F+1) <> "C"
                THEN ST=F+1:GOTO 120
        140 PRINT "FOUND 'AB' OR 'AC'"

This illustrates the importance of the aexp's use as a starting position

6.3.4 LEFT$

       Format: LEFT$(svar, aexp)

       Example: 100 A$=LEFT$("ABCDE",3)
                    200 PRINT LEFT$("ABCD",5)

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

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

6.3.5 LEN

       Format: LEN(sexp)

       Example: 100 PRINT LEN(A$)

This function returns the length in bytes of the designated string. This information may then be printed or used later in a program. The length of a string variable is simply the index for the character which is currently at the end of the string. Strings have a length of 0 until characters have been stored in them. It is possible to store into the middle of the string by using subscripting. However, the beginning of the string will contain garbage.

The following routine illustrates one use of the LEN function:

```
10 A$="ATARI"            10 DIM AR$(3,0)
20 PRINT LEN(A$)         20 AR$(2;)="ATARI"
                         30 PRINT LEN(AR$(2;))
```

The result of running either of the above programs would be 5.

6.3.6 MID$

    Format: MID$(svar,aexp1,aexp2)

    Example: A$=MID$("ABCDEFG",2,4)

MID$ allows you to get a substring from the middle of another string. The substring starts at the 'aexp1' character of svar, and is 'aexp2' characters long.

If aexp1 equals 0 an error occurs (since there is no zeroeth character of a string), but if aexp1 is greater than the length of svar no error occurs (and no characters are returned).

Aexp2 is allowed any positive number (including 0), but if its value makes the substring go beyond the length of svar, then the substring returned ends at the end of svar.

6.3.7 RIGHT$

    Format: RIGHT$(svar,aexp)

    Example: A$=RIGHT$("123456",4)

This function is used to return the rightmost 'aexp' characters of 'svar'. If aexp is greater than the number of characters in svar, then the entire string 'svar' is returned.

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

6.3.8 STR$

    Format: STR(aexp)

    Example: A$=STR$(65)

This function returns the string form of the number in parentheses. The above example would return the actual number 65, but it would be recognized by the computer as a string.

NOTE: There can only be one STR$ and only one CHR$ in a logical comparison. For example, A=STR$(1)>STR$(2) is not valid and will not work correctly.

6.3.9 VAL

Format: VAL(sexp)

Example: 100 A=VAL(A$)

This function is the opposite of the STR$ function, in that it returns the number represented by a string, providing that the string is indeed a string representation of a number. Using this function, the computer can perform arithmetic operations on strings as shown in the following example program:

```
10 DIM B$(5)
20 B$="10000"
30 B=SQR(VAL(B$))
40 PRINT "THE SQUARE ROOT OF ";B$;" IS ";B
```

Upon execution, the screen displays:

THE SQUARE ROOT OF 10000 IS 100.

It is not possible to use the VAL function with a string that does not start with a number, or that cannot be interpreted by the computer as a number. It can, however, interpret floating point numbers (e.g., VAL("1E9") would return the number 1000000000).

6.3.10 An Example Program

The following program inputs a three word string, cuts it up into the separate words through the use of LEFT$, MID$, and RIGHT$, and then prints out the ATASCII value of each letter in each word using ASC. Note that this program also uses the LEN and FIND functions.

```
100 PRINT "Give me a three word string with each"
110 INPUT "word separated by a space> ",S$
120 POS1=FIND(S$," ",0) : REM find end of 1st word
130 L$=LEFT$(S$,POS1-1) : REM fill 1st word string
140 POS2=FIND(S$," ", POS1) : REM find 2nd word
150 M$=MID$(S$,POS1+1, POS2-POS1-1) : REM fill 2nd word string
160 R$=RIGHT$(S$,LEN(S$)-POS2) : REM fill 3rd word string
170 PRINT "*** ";L$ : REM print 1st word
180   FOR I=1 TO LEN(L$) : REM print ASC value of each letter
190   PRINT ,L$(I,I); ": "; ASC(L$(I))
200   NEXT I
210 PRINT "*** ";M$ : REM print 2nd word
220   FOR I=1 TO LEN(M$) : REM print ASC value of each letter
230   PRINT ,M$(I,I); ": "; ASC(M$(I))
240   NEXT I
250 PRINT "*** ";R$ :  REM print 3rd word
260   FOR I=1 TO LEN(R$) : REM print ASC value of each letter
270   PRINT ,R$(I,I); ": "; ASC(R$(I))
280 NEXT I
290 GOTO 100
```

NOTE: line 130, 150, and 160 could have been coded as follows:

```
        130 L$=S$(1,POS1-1)
        150 M$=S$(POS1+1,POS2-1)
        160 R$=S$(POS2+1)
```

6.4 Game Controller Functions

6.4.1 HSTICK

      Format: HSTICK(aexp)

      Example: 100 IF HSTICK(0)>0 THEN PRINT "MOVE RIGHT"

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 – 3), and the values returned (and their meanings) are as follows:

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

6.4.2 PADDLE

      Format: PADDLE(aexp)

      Example: PRINT PADDLE(3)

This function returns the current value of a particular paddle. aexp is the number of the paddle port (0-7). The value returned will be between 1 and 228, with the number increasing as the knob is turned counterclockwise.

6.4.3 PEN

      Format: PEN(aexp)

      Example: PRINT "light pen at X="; PEN(0)

The PEN function simply reads the ATARI light pen registers and returns their contents to the user. The number specified by aexp is interpreted as follows:

      PEN(0) reads the horizontal position register
      PEN(1) reads the vertical position register

6.4.4 PTRIG

      Format: PTRIG(aexp)

      Example: 100 IF PTRIG(1)=0 THEN PRINT "MISSILES
                               FIRED!"

The PTRIG function returns the status of 0 if the trigger button of the designated paddle is pressed. Otherwise, it returns a value of 1. The aexp must be a number between 0 and 7 as it designates the paddle.

6.4.5 STICK

      Format: STICK(aexp)

      Example: 100 PRINT STICK(3)

This function works exactly the same way as PADDLE command, but is used with the joystick controllers. aexp is the number of the joystick port (0 – 3). The following diagram shows the values returned by this function:

10	14	6
11	15	7
9	13	5

COMMENT: This function was the only means given to access the joystick with original Atari BASIC. For most purposes, HSTICK and VSTICK are much easier to use and to work with.

6.4.6 STRIG

      Format: STRIG(aexp)

      Example: 100 IF STRIG(1)=0 THEN PRINT "FIRE TORPEDO"

The STRIG function works the same way as the PTRIG function, except that it is used with the joysticks instead of the paddles.

6.4.7 VSTICK

      Format: VSTICK(aexp)

      Example: IF VSTICK(0)<0 THEN PRINT "MOVE DOWN"

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 – 3), and the values returned (and their meanings) are as follows:

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

6.4.8 An Example Program

The following program creates a simple GRAPHICS mode 5 sketchpad using the game controller functions HSTICK, VSTICK, and STRIG to move and draw.

```
100 GRAPHICS 5 : REM set up screen
110 COL=40 : REM middle of screen
120 ROW=20
130 COLOR 2 : REM drawing a cursor color
140 PLOT COL, ROW : REM plot cursor
150    FOR I=1 TO 15 : NEXT I : REM delay loop
160 IF STRIG(0)=1 THEN COLOR 0 : PLOT COL, ROW : REM
            don't draw point
170 COL=COL+HSTICK(0) : REM check for movement
180 ROW=ROW-VSTICK(0)
190 IF COL<0 THEN COL=0 : REM screen bounds checking
200 IF COL>79 THEN COL=79
210 IF ROW<0 THEN ROW=0
220 IF ROW>39 THEN ROW=39
230    FOR I=1 TO 25 : NEXT I : REM delay loop
240 GOTO 130 : REM repeat
```

6.5 Player/Missile Functions

For examples showing the use of the P/M functions, see section 8.13.

6.5.1 BUMP

        Format: BUMP(pmnum,aexp);

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

BUMP accesses the 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 (the aexp) may be either a player number or playfield number (see section 8.2 for the appropriate number).

Valids BUMPs: PLAYER to PLAYER (0-3 to 0-3)
                              MISSILE to PLAYER (4-7 to 0-3)
                              PLAYER to PLAYFIELD (0-3 to 8-11)
                              MISSILE to PLAYFIELD (4-7 to 8-11)

NOTE: BUMP (p,p), where the p's are 0 through 3 and identical, always returns a 0.

NOTE: It is advisable to 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 by using the following statement:

      POKE 53278,0

6.5.2 PMADR

        Format: PMADR(aexp)

        Examples: P0=PMADR(0)

This function may be used in any arithmetic expression and is used to obtain 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. (See section 8.13 for examples of its use, and section 8.2 for a description of the aexp values.)

NOTE: PMADR(m) – where m is a missile number (4 through 7) returns the same address for all missiles.

6.6 Special Purpose Functions

6.6.1 ADR

      Format: ADR(svar)

      Example: ADR(A$)
                    ADR(B$(5;))

Returns the decimal memory address of the string specified by the expression in parentheses. Knowing the address enables the programmers to pass the information to USR routines, etc. (See USR and Appendix D).

6.6.2 DPEEK

      Format: DPEEK(aexp)

      Example: PRINT "variable table is at ";:DPEEK(130)

The DPEEK function is very similar to the PEEK function, except that it allows you to look at two consecutive bytes of information. This is especially useful when looking at two byte locations containing address information, as in the above example. If you did this example using PEEKs, it would look like:

      PRINT "variable name table is at ";
      PRINT PEEK(130)+(PEEK(131)*256)

It is easy to see that using DPEEK is much easier.

6.6.3 DPOKE

      Format: DPOKE aexp1,aexp2

      Example: DPOKE 88,32768

DPOKE is similar to POKE, except that it allows you to put two bytes of data into memory instead of one. aexp1 is the address where you want the data to go, and aexp2 is the data itself. 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 32768. to do this using POKEs, you would need to do an amazing amount of math to get the right number into each of the two bytes.

6.6.4 ERR

Format: ERR(aexp)

Example: PRINT "ERROR ";ERR(0); " OCCURRED AT LINE "
                        ;ERR(1)

This function – in conjunction with TRAP, CONT, and GOTO allows the BASIC XL programmer to effectively diagnose and dispatch virtually and run-time error.

ERR(0)   returns the last run-time error number
ERR(1)   returns the line number where the error occurred

Example: 100 TRAP 200
                110 INPUT "A NUMBER, PLEASE >>",NUM
                120 PRINT "A VALID NUMBER" : END
                200 IF ERR(0)=8 THEN GOTO ERR(1)
                210 PRINT "UNEXPECTED ERROR #";ERR(0)

6.6.5 FRE

Format: FRE(aexp)

Example: PRINT FRE(0)
                100 IF FRE(0)<1000 THEN
                PRINT "MEMORY CRITICAL"

This function returns the number of bytes of user RAM left. Its primary use is in Direct mode with a dummy variable (0) to inform the programmer how much memory space remains for completion of a program. Of course FRE can also be used within a BASIC program in Deferred mode.

6.6.6 HEX$

Format: HEX$(aexp)

Example: 100 PRINT HEX$(X+7)
                200 A$=HEX$(83)
                210 PRINT "$";A$(3,4)

This function will convert aexp to a four digit hexadecimal humber.

The second example shows how you can obtain a two digit hex number for printing or other manipulation.

NOTE: No "$" is place din front of the number.

6.6.7 PEEK

        Format: PEEK(aexp)

        Example: 1000 IF PEEK (4000) = 255 THEN PRINT "255"
                   100 PRINT "LEFT MARGIN IS":PEEK(82)

Returns the contents of a specified memory address location (aexp). The address specified must be an integer or an arithmetic expression that evaluates to an integer between 0 and 65535 and represents the memory address in decimal notation (not hexadecimal). The number returned will also be a decimal integer with a range from 0 to 255. this function allows the user to examine either RAM or ROM location. In the first example above, the PEEK is used to determine whether location 4000 (decimal) contains the number 255. In the second example, the PEEK function is used to examine the left margin.

6.6.8 POKE

        Format: POKE aexp1,aexp2

        Example: POKE 82,10
                   100 POKE 82,20

Although this is not a function, it is included in this section because it is closely associated with the PEEK memory location or modifies data already stored there. In the above format, aexp1 is the decimal address of the location to be POKEd and aexp2 is the data to be POKEd. Note that this number is a decimal number between 0 and 255. POKE cannot be used to alter ROM locations. In gaining familiarity with this command is advisable to look at the memory location with a PEEK and write down the contents of the location. Then, if the POKE doesn't work as anticipated, the original contents can be poked back into the location.

The above Direct mode example changes the left screen margin from its default position of 2 to a new position of 10. In other words, the new margin will be 8 spaces to the right. To restore the margin to its normal default position, press <SYSTEM RESET>.

6.6.9 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 given BASIC XL system function. These system functions can be changed using the SET command, and SYS allows you to find out what any current value is. aexp is the number of the system function as defined in the SET section (3.15).

6.6.10 TAB

        Format: TAB(aexp)

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

The TAB function's effect is identical with that of the TAB statement (section 5.28). The difference is that, for PRINT USING statements, an imbedded TAB function simplifies the programmers task greatly.

TAB will output ATASCII space characters to the current PRINT file or device (#3 in our example). Sufficient space will be output so that the next item will print in the column specified (only if TAB is followed by a semi-colon, though). If the column specified is less than the current column, a RETURN will be output first.

CAUTION: The TAB function will output spaces on some device whenever it is used; therefore, it should be used ONLY in PRINT statements.

6.6.11 USR

> Format: USR(aexp1 [,aexp2][aexp3…])
>
> Example: 100 RESUKT = USR (ADD1,A*2)

This function returns the results of a machine-language subroutine. The first expression, aexp1, must be an integer or arithmetic expression that evaluates to an integer that represents the decimal memory address of the machine language routine to be performed. The input arguments aexp2,aexp3,etc., are optional. These should be arithmetic expressions within a decimal range of 0 through 65535. A non-integer value may be used; however, it will be rounded to the nearest integer.

These values will be converted from BASIC's Binary Coded Decimal (BCD) floating point number format to a two-byte binary number, then pushed onto the hardware stack.

The arguments are pushed in the reverse of the order given, so the assembly language program may then pull them in proper forward order. Additionally, the one-byte count of parameters is pushed onto the stack and MUST be popped by the USeR routine (except see section 3.15, the SET command).

Also, if all arguments are properly pulled from the stack, then the USeR routine may return to BASIC XL by simply executing an RTS instruction. And, finally, the routine may return a single 16-bit value to BASIC XL(as the "value" of the USeR function) by placing a result in FR0 and FR0+1 ($D4 and $D5) before returning.

Example: The following example uses a USR call to XOR two numbers
           (the arguments to the USR routine) and then return that value
           to BASIC XL.

USR routine at $680:
```
FR0     =     $D4
        *=    $680
        PLA                 : get number of arguments
        CMP     #2          : see if it's 2
        BNE     *           : loop forever if wrong num. of args.
        PLA                 : get high byte of arg #1
        STA     FR0+1       : store high byte
        PLA                 : get low byte of arg #1
        STA     FR0         : store low byte
        PLA                 : get high byte of arg #2
        EOR     FRO+1       : XOR it with high byte of arg #1
        STA     FRO+1       : store result of XOR
        PLA                 : get low byte of arg #2
        EOR     FR0         : XOR it with low byte of arg #1
        STA     FR0         : store result of XOR
        RTS                 : end of USR routine
```

6.6.12 An Example Program

The following program uses the system timer located at $12, $13, and $14 to create a countdown clock. This is done by poking 0 into the low byte of the timer and waiting until it is greater than or equal to 60.

```
100 GRAPHICS 2
110 PRINT #6; CHR$(125) : REM Clear Mode 2 area
120 PRINT : PRINT : PRINT
130 PRINT "COUNTDOWN TIME? ";
140 INPUT #0,X
150 POKE $14,0 : REM set clock = 0
160 PRINT #6; "TIME – ";
170   WHILE X>0 : REM start the countdown
180   POSITION 7,1 : REM get ready to print the new time
190   PRINT #6; USING "##",X; : REM print time left
200     WHILE PEEK($14)<=60 : REM wait until a second has passed
210     ENDWHILE
220   POKE $14,0 : REM reset the clock for the next second
230   X=X-1 : REM decrement number of seconds left
240   ENDWHILE : REM end of countdown loop
250 PRINT CHR$(253) : REM ring the bell
260 GOTO 110 : REM do the whole thing over again
```

This chapter describes the BASIC XL commands used to manipulate the wide variety of screen graphics available on the Atari personal computers. It also describes the BASIC XL command used to manipulate the sound generating mechanism of the Atari computers.

7.1 GRAPHICS (GR.)

> Format: GRAPHICS aexp
>
> Example: GRAPHICS 2

This command is used to select one of the nine graphics modes. The tables below summarizes the nine modes and the characteristics of each.

The GRAPHICS command 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.

NOTE: aexp must be positive.

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

To return to graphics mode 0 in direct mode, press <SYSTEM RESET> or type GR.0 and press <RETURN>.

Gr. Mode	Mode Type	Cols	(split) Rows	(full) Rows	Num of Colors
0	TEXT	40	N/A	24	2
1	TEXT	20	20	24	5
2	TEXT	20	10	12	5
3	GRAPHICS	40	20	24	4
4	GRAPHICS	80	40	48	2
5	GRAPHICS	80	40	48	4
6	GRAPHICS	160	80	96	2
7	GRAPHICS	160	80	96	4
8	GRAPHICS	320	160	192	1 ½
9	GRAPHICS	80	N/A	192	16
10	GRAPHICS	80	N/A	192	9
11	GRAPHICS	80	N/A	192	16

7.1.1 GRAPHICS Mode 0

This mode is the 1-color, 2-luminance(brightness) default mode for the ATARI personal computer. It contains a 24 line by 40 character screen matrix. The default margin settings at 2 and 39 allow 38 characters per line. Margins may be changed by poking 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. This full-screen display has a blue display area bordered in black (unless the border is specified to be another color). To display characters at a specified location, use one of the following methods:

        POSITION aexp1,aexp2 : REM Puts cursor at location
        PRINT sexp : specified by aexp1 and aexp2.

GRAPHICS 0 is also used as a clear screen command either in Direct mode or Deferred mode. It terminates any previously selected graphics mode and returns the screen to the default mode (GRAPHIC 0).

7.1.2 GRAPHICS Modes 1 and 2

These two 5-color modes are Text modes. However, they are both split-screen modes.

Characters printed in Graphics mode 1 are twice the width of those printed in Graphics mode 0, but are the same height.

Characters printed in Graphics mode 2 are twice the width and height of those in Graphics mode 0.

In the split-screen mode, a PRINT command is used to display characters in either the text window or the graphic window. To print characters in the graphic window, specify channel #6 after the PRINT command.

> Example: 100 GR. 1
> 110 PRINT #6;"A MODE 1 TEST"

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

Character Type	Color Register	Default Color
Upper case alphabetic	0	Orange
Lower case alphabetic	1	Light Green
Inverse upper case alphabetic	2	Dark Blue
Inverse lower case alphabetic	3	Red
Numbers	0	Orange
Inverse Numbers	2	Dark Blue

NOTE: See SETCOLOR to change character colors:

Unless otherwise specified, all characters are displayed in upper case non-inverse form. To print lower case letters and graphics characters, use a POKE 756,226. To return to upper case, use POKE 756, 224.

In graphics modes 1 and 2, there is no inverse video, but it is possible to get all the rest of the characters in four different colors (see end of section).

7.1.3 GRAPHICS Modes 3.5. and 7

These three 4-color graphics 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 are alike except that modes 5 and 7 use more points (pixels) in plotting, drawing, and positioning the cursor; the points are smaller, thereby giving a much higher resolution.

7.1.5 GRAPHICS Mode 8

This graphics mode gives the highest resolution of all the other modes. 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 luminance's, as mode 0.

7.1.6 GRAPHICS modes 9, 10, and 11

GRAPHICS modes 9, 10, and 11 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 modes. 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. 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 = 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. Below is a table showing how the pixel values relate to the color registers and what BASIC XL command may be used.

VALUE	REGISTER	REG. ADDRESS	COMMAND
0	PCOLR0	704	PMCOLOR 0
1	PCOLR1	705	PMCOLOR 1
2	PCOLR2	706	PMCOLOR 2
3	PCOLR3	707	PMCOLOR 3
4	PCOLR4	708	SETCOLOR 0
5	PCOLR5	709	SETCOLOR 1
6	PCOLR6	710	SETCOLOR 2
7	PCOLR7	711	SETCOLOR 3
8	PCOLR8	712	SETCOLOR 4

7.2 COLOR (C.)

      Format: COLOR aexp

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

The value of the expression in the COLOR statement determines the data to be stored in the display memory for all subsequent PLOT and DRAWTO commands until the next COLOR statement is executed. The value must be positive and is usually an integer from 0 through 255. Non-integers are rounded to the nearest integer. The graphics display hardware interprets this data in different ways in the different graphics modes.

In text modes 0 through 2, the number can be from 0 through 255 (8 bits) and determines the character to be displayed and its color. (The two most significant bits determines the color. This is why only 64 different characters are available in these modes instead of the full 256-character set.)

Graphics modes 3 through 8 are not text modes, so the data stored in the display RAM simply determines the color of each pixel. Two-color or two-luminance modes require either 0 or 1 (1-bit) and four-color modes require 0, 1, 2, or 3. (The expression in the COLOR statement may have a value greater than 3, but only one or two bits will be used.)

The actual color which is displayed depends on the value in the color register which corresponds to the data of 0, 1, 2, or 3 in the particular graphics mode being used. This may be determined by looking in the table at the end of the SETCOLOR section. This table gives COLOR and SETCOLOR relationships for all the GRAPHICS modes.

Note that when BASIC XL is first powered up, the color data is 0, and when graphics command (without +32) is executed, all of the pixels are set to 0. Therefore, nothing seems to happen to PLOT and DRAWTO in GRAPHICS 3 through 7 when no COLOR statement has been executed. Correct this by doing a COLOR 1 first.

7.3 DRAWTO (DR.)

      Format: DRAWTO aexp1,aexp2

      Example: 100 DRAWTO 10,0

This statement causes a line to be drawn from the last point displayed by a PLOT (see PLOT) to the location by aexp1 and aexp2. The first expression represents the X coordinate (column) and the second represents the Y coordinate (row). The color of the line is the same color as the point displayed by the PLOT.

7.4 LOCATE (LOC.)

Format: LOCATE aexp1,aexp2,avar

Example: 150 LOCATE 11,15,X

This command positions the invisible graphics cursor at the specified location in the graphics window, retrieves the data at that pixel, and stores it in the specified arithmetic variable. This gives a number from 0 to 255 for Graphics modes 0 through 2, a 0 or 1 for the 2-color graphics modes, and a 0, 1, 2, or 3 for the 4-color modes. The two arithmetic expressions specify the X and Y coordinates of the point. LOCATE is equivalent to:

POSITION aexp1,aexp2:GET #6,avar

Doing a PRINT after a LOCATE or GET from the screen may cause the data in the pixel which was examined to be modified. This problem is avoided by repositioning the cursor and putting the data that was read back into the pixel before doing the PRINT. The following program illustrates the use of the LOCATE command:

```
10 GRAPHICS 3+16
20 COLOR 1
30 SETCOLOR 2,10,8
40 PLOT 10,15
50 DRAWTO 15,15
60 LOCATE 12,15,X
70 PRINT X
```

On execution, the program prints the data (1) determined by the COLOR statement which was stored in pixel 12,15.

7.5 PLOT (PL.)

Format: PLOT aexp1,aexp2

Example: 100 PLOT 5,5

The PLOT command is used in graphics modes 3 through 8 to display a point in the graphics window. aexp1 specifies the X-coordinate and aexp2 specifies the Y-coordinate. The color of the plotted point is determined by the due and luminance in the color register from the last COLOR statement executed. To change this color register, and the color of the plotted point, use SETCOLOR. Points that can be plotted on the screen are dependent on the graphics mode being used. The range of points begins at (0,0), and extends to one less than the total number of rows (X-coordinate) or columns (Y-corrdinate).

NOTE: PLOT aexp1,aexp2 is equivalent to:
         POSITION aexp1,aexp2 : PUT #6, COLOR

7.6 POSITION (POS.)

        Format: POSITION aexp1,aexp2

        Example: 100 POSITION 8,12

The POSITION statement is used to place the invisible graphics window cursor at the specified location on the screen (usually precedes a PRINT or PUT statement). This statement can be used in all modes. Note that the cursor does not actually move until an I/O command which involves the screen is issued.

7.7 PUT and GET (as applied to graphics)

        Format: PUT #6,aexp
                    GET #6,avar

        Example: 100 PUT #6,ASC("A")
                    200 GET #6,X

In graphics work, PUT is used to output data to the screen display. This statement works hand-in-hand with the POSITION statement. After a PUT (or GET), the cursor is moved to that next location on the screen.

Doing a PUT to device #6 causes the one-byte aexp to be displayed at the cursor position. The byte is either an ATASCII code byte for a particular character (modes 0-2) or the color data (modes 3-8).

GET is used to input the code byte of the character displayed at the cursor position, into the specified arithmetic variable. The values used in PUT and GET correspond to the values in the COLOR statement. (PRINT and INPUT may also be used.)

NOTE: Doing a PRINT after a LOCATE or GET from the screen may cause the data in the pixel which was examined to be modified. To avoid this problem, reposition the cursor and put the data that was read back into the pixel before doing the PRINT.

7.8 SETCOLOR (SE.)

       Format: SETCOLOR aexp1,aexp2,aexp3

       Example: 100 SETCOLOR 0,1,4

This statement is used to choose the particular hue and luminance to be stored in the specified color register. The parameters of the SETCOLOR statement are defined below:

aexp1 = Color register (0-4 depending on graphics mode)
aexp2 = Color hue number (0-15 – see the table below)
aexp3 = Color luminance (must be an even number between 0 and 14; The
       higher the number, the brighter the display. 14 is almost pure white.)

SETCOLOR aexp2	Color	SETCOLOR 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	Purple	13	Yellow-Green
6	Purple-Blue	14	Orange-Green
7	blue	15	Light Orange

NOTE: Colors will vary with type and adjustment of TV or monitor used.

The ATARI display hardware contains five color registers, numbered from 0 through 4. The Operating System (OS) has five RAM locations (COLOR0 through COLOR4, see Appendix I – Memory Locations) where it keeps track of the current colors. The SETCOLOR statement is used to change the values in these RAM locations. (The OS transfers these values to the hardware registers every television frame.)

The SETCOLOR statement requires a value from 0 to 4 to specify a color register. The COLOR statement uses different numbers because it specifies data which only indirectly corresponds to a color register. This can be confusing, so careful study of the various tables in this section is advised.

SETCOLOR Register	Default Color	Default Luminance	Color
0	2	8	Orange
1	12	10	Green
2	9	4	Dark Blue
3	4	6	Pink or Red
4	0	0	Black

"DEFAULT" occurs if no SETCOLOR statement is used.

The following table shows the COLOR – SETCOLOR relationships for all the GRAPHICS modes, and gives some information on the registers used in a specific mode:

GRAPHICS Mode	SETCOLOR Register	COLOR Number	Description and Comments
0 and all text windows	0	COLOR data actually determines	--
	1		--
	2		Character luminance
	3		Background
	4		Border
1,2	0	The character to PLOT	Character
	1		Character
	2		Character
	3		Character
	4		Background, Border
3,5,7	0	1	Graphics Point
	1	2	Graphics Point
	2	3	Graphics Point
	3	--	--
	4	0	Gr. Pt., Border, Background
4,6	0	1	Graphics Point
	4	0	Gr. Pt., Border, Background
8	1	1	Graphic Point luminance
	2	0	Gr. Pt, Background
	4	--	Border

7.9 XIO (X.) Special Fill Application

>   Format: XIO 18,#aexp,aexp1,aexp2,filespec

>   Example: 100 XIO 18,#6,0,0,"S:"

This special application of the XIO statement fills an area on the screen between plotted points and lines with a non-zero color value. Dummy variables (0) are used for aexp1 and aexp2.

The following steps illustrate the fill process:

>   1.   PLOT bottom right corner (point 1).
>   2.   DRAWTO upper right corner (point 2). This outlines the right edge of the area to be filled.
>   3.   DRAWTO upper left corner (point 3).
>   4.   POSITION cursor at lower left corner (point 4).
>   5.   POKE address 765 with the fill color data (1,2, or 3).

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 fill cannot be used to change an area which has been filled in with a non-zero value, as the fill will stop.

WARNING: The fill command will go into an infinite loop if you attempt to put zero (0) data on a line which has no non-zero pixels. <BREAK> or <SYSTEM RESET> can be used to stop the fill if this happens.

The following program creates a shape and fills it with a data (color) of 3. Note that the XIO command draws in the lines of the left and bottom of the figure.

```
10 GRAPHICS 5+16
20 COLOR 3
30 PLOT 70,45
40 DRAWTO 50,10
50 DRAWTO 30,10
60 POSITION 10,45
70 POKE 765,3
80 XIO 18,#6,0,0,"S:"
90 GOTO 90
```

7.10 SOUND (SO.)

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 statement with the same aexp1 or an END statement. The SOUND parameters are described as follows:

aexp1   Is one of the four voice available on the Atari (number 0 – 3).

aexp2   Is the frequency (pitch) of the sound, and ranges between 0 and 255. 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:

	aexp2	Note(s)		aexp2	Note(s)
HIGH	29	C		91	F
NOTES	31	B		96	E
	33	A# or Bb		102	D# or Eb
	35	a		108	D
	37	G# or Ab		114	C# or Db
	40	G	MIDDLE C	121	C
	42	F# or Gb		128	B
	45	F		136	A# or Bb
	47	E		144	A
	50	D# or Eb		153	G# or Ab
	53	D		162	G
	57	C# or Db		173	F# or Gb
	60	C		182	F
	64	B	LOW	193	E
	68	A# or Bb	NOTES	204	D# or Eb
	72	A		217	D
	76	G# or Ab		230	C# or Db
	81	G		243	C
	85	F# or Gb			

The following program plays a C scale using the above values:

```
10 READ A
20 IF A=256 THEN END
30 SOUND 0,A,10,10
40 FOR W=1 TO 400:NEXT W
50 PRINT A
60 GOTO 10
70 END
80 DATA 29,31,35,40,45,47,53,60,64,72,81,91,96,100,121
90 DATA 120,144,162,182,193,217,243,256
```

Note 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.

This chapter describes the BASIC XL commands and functions used to access the Atari's Player-Missile Graphics. Player Missile Graphics (hereafter usually referred to as simply "PMG") represents 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 XL goes a long way toward remedying these omissions by adding six PMG commands (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.

For information on the PMG functions, see section 6.5.

8.1 An Overview of P/M Graphics

For a complete technical discussion of PMG, and to learn of even more PMG "tricks" that are included in BASIC XL, read the Atari document entitled "Atari 400/800 Hardware Manual" (Atari part number C0165555, Rev. 1 or later).

It was 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 selected and any COLOR or SETCOLOR commands currently active. In Atari (and now BASIC XL) parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical strip on the screen. Sounds dull? Consider: each player (there are four) may be "painted" in any of 128 colors available on the Atari (see SETCOLOR for specific colors). Within the vertical strip, 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 command in BASIC XL). We may move the player vertically by simply doing a circular shift on the contiguous memory block representing the player (again, the PMMOVE command of BASIC XL 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 locations on the screen. To complicate:

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

1) They are only 2 bits wide, and all four missiles 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 XL commands (MISSILE and PMMOVE, for example), you the programmer need not be too aware of the mechanisms of PMG.
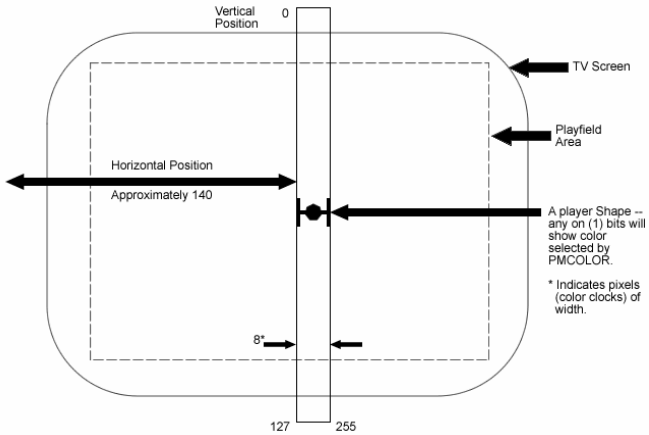
8.2 P/M Graphics Conventions

1. Players are numbered from 0 through 3. Each player has a corresponding missile whose number is 4 greater than that of its parent player, thus missiles are numbered 4 through 7. In the BUMP function, the "playfields" are numbered from 8 through 11, corresponding to actual playfields 0 through 3. (Note: Playfields are actually COLORs on the main GRAPHICS screen, and can be PLOTted, PRINTed, etc).

2. There is some inconsistency in which way is "UP". PLOT, DRAWTO, POKE, MOVE, 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. [If this really bothers you please let us know!].

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).

8.3 BGET and BPUT with P/M's

As with MOVE (see section 8.11), 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!

Example: BGET #3, PMADR(0),128

Would get a PMG.2 mode player from the file opened in slot #3.

Example: BGET #4,PMADR(4),256*5

Would fill all the missiles AND players in PMG.1 mode – with a single statement!

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

8.4 PMCLR

Format: PMCLR pmnum

Example: PMCLR 4

This statement "clears" a player or missile area to all zero bytes, thus "erasing" the player/missile. PMCLR is aware of what PMG mode is active and clears only the appropriate amounts of memory. CAUTION: PMCLR 4 through PMCLR 7 all produce the same action – ALL missiles are cleared, not just the one specified. To clear a single missile, try the following:

SET 7,0 : PMMOVE 4;255

8.5 PMCOLOR (PMCO.)

        Format: PMCOLOR pmnum,aexp,aexp

        Example: PMCOLOR 2,13,8

PMCOLORs are identical in usage to those of the SETCOLOR statement except that a player/missile set has its color chosen. 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 13).

NOTE: PMG has NO default colors set on power-up or SYSTEM RESET.

8.6 PMGRAPHICS (PMG.)

        Format: PMGRAPHICS aexp

        Example: PMG. 2

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

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

Single and Double line resolution (hereafter referred to as "PMG Modes") refer to the height which a byte in the player "stripe" occupies – either one or two television scan lines. (A scan line height is the pixel height in GRAPHICS mode 8. GRAPHICS 7 has pixels 2 scan lines high, similar to PMG.2)

The secondary implication of single line versus double line resolution is that single line resolution requires twice as much memory as double line, 256 bytes per player versus 128 bytes. The following diagram shows PMG memory usage in BASIC XL, but the user really need not be aware of the mechanics if the PMADR function is used.

RAMSZ ($6A)

```
┌─────────────────────────────────────────┐
│                                          │
│          Current GRAPHICS Mode           │
│                                          │
└─────────────────────────────────────────┘
```

Depending on GRAPHICS mode, there may or may not be unused memory here.

	Double Line				Single Line		
+1024							+2048
+856	Player 3				Player 3		
+708	Player 2						+1792
+512	Player 1				Player 2		
+384	Player 0						+1536
	M3	M2	M1	M0	Player 1		
							+1280
					Player 0		
PMBASE							+1024
					M1	M2	M3	M4	+768
							PMBASE

PMBASE

NOTE: MEMTOP ($2E5) points to the bottom of the missiles (PMBASE+384 in double line, PMBASE+768 in single line.

8.7 PMMOVE

        Format: PMMOVE pmnum[,aexp][;aexp]

        Example: PMMOVE 0,120;1
                  PMMOVE 1,80
                  PMMOVE 4;-3

Once a player or missile has been "defined" (via POKE, MOVE, GET, or MISSILE), the truly unique features of PMG under BASIC XL may be utilized. With PMMOVE, the user may position the player/missile shape anywhere on the screen almost instantly.

BASIC XL allows the user to position each player and missile independently. Because of the hardware implementation, though, there is a difference in how horizontal and vertical positions are specified.

The parameter following the comma in PMMOVE 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 the specification of the LEFT edge: changing a player's width (see PMWIDTH) will not change the position of its left edge, but will expand the player to the right.

The parameter following the semicolon in PMMOVE 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 – either towards higher memory (down the screen) or lower memory (up the screen). BASIC XL allows the user to specify a vertical movement from -255 (down 255 pixels) to +255 (up 255 pixels).

NOTE: The +/- convention on vertical movement conforms to the value returned by VSTICK.

        Example: PMMOVE N;VSTICK(N)

Will move player N up or down (or not move him) in accordance with the joystick position.

NOTE: SET may be used to tell PMMOVE whether an object should "wraparound" (from bottom of the screen to top of the screen or vice versa) or should disappear as it scrolls to far up or down. SET 7,1 specifies wraparound, and SET 7,0 disables it.

8.8 PMWIDTH (PMW.)

        Format: PMWIDTH pmnum,aexp

        Example: PMWIDTH 1,2

Just as PMGRAPHICs can select single or double pixel heights, PMWIDTH
allows the user to specify the screen width of players and missiles. But where
PMGRAPHICs selects resolution mode for all players and missiles,
PMWIDTH allows each player AND missile to be separately specified. The
aexp used for the width should have values of 1,2, or 4 – representing the
number of color clocks (equivalent to a pixel width in GRAPHICS mode 7)
which each bit in a player definition will occupy.

NOTE: PMG.2 and PMWIDTH 1 combine to allow each bit of a player
definition to be equivalent to a GRAPHICS mode 7 pixel – a not altogether
accidental occurrence.

NOTE: Although players may be made wider with PMWIDTH, the resolution
then suffers. Wider "players" may be made by placing two or more separate
players side-by-side.

8.9 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.

Example: 100 FOR LOC=48 TO 52
            110 READ N : POKE LOC+PMADR(0),N
            120 NEXT LOC
            …
            900 DATA 255,129,255,129,255

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

8.10 MISSILE (MIS.)

        Format: MISSILE pmnum,aexp,aexp

        Example: MISSILE 4,48,3

The MISSILE statement allows an easy way for a parent player to "shoot" a missile. The first aexp specifies the absolute vertical position of the beginning of the missile (0 is the top of the screen), and the second aexp specifies the vertical height of the missile.

Example: MISSILE 4,64,3

Would place a missile 3 or 6 scan lines high (depends on PMG. Mode) 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 with the current missile memory. This can allow the user to erase existing missiles while creating others.

Example: MISSILE 5,40,4
               MISSILE 5,40,8

The first statement creates a 4 pixel missile at vertical position 20. The second statement erases the first missile and creates a 4 pixel missile at vertical position 24.

8.11 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 MOVEd into place at will.

Example: MOVE ADR(A$), PMADR(2),128

        Could move an entire double line resolution player from A$ to player stripe number 2.

        POKE PMADR(1),255 : MOVE PMADR(1),PMADR(1)+1,127

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

8.12 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 easily interfaced to BASIC XL.

Examples: A=USR(PMBLINK,PMADR(2),128)

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

8.13 Example PMG Programs

1. A very simple program with one player and its missile.

```
100 SETCOLOR 2,0,0          : REM note we leave ourselves in GR.0
110 PMGRAPHICS 2            : REM double line resolution
120 LET width=1 : y=48      : REM just initializing
130 PMCLR 0 : PMCLR 4       : REM clear player 0 and missile 0
140 p=PMADR(0)              : REM gets address of player
150   FOR i=p+y to p+y+4    : REM a 5 element player to be defined
160   READ val              : REM see below for DATA scheme
170   POKE i,val            : REM actually setting up player shape
180   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 to make 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 again at top of player
300   FOR x=127 to 255      : REM the missile movement loop
310   PMMOVE 4,x            : REM moves missile 0
320   SOUND 0,255-x,10,15
330     IF (x & 7) = 7      : REM every eighth horizontal position
340       MISSILE 0,y,5     : REM you have to see this to believe it
350     ENDIF               : REM could have had an else, of course
360   NEXT x
370 PMMOVE 0,0              : REM so width doesn't change on screen
400 width=width*2           : REM we will make the player wider
410 IF width > 4 THEN width = 1: REM until it gets too wide
420 PMWIDTH 0,width         : REM the new width
430 PMCLR 4                 : no more missile
440 GOTO 200
450 REM
500 REM ********** THE DATA FOR PLAYER SHAPE **********
510 REM
520 DATA 151                : REM $99      *  **   *
530 DATA 189                : REM $BD      * ****  *
540 DATA 255                : REM $FF      ********
550 DATA 189                : REM $BD      * ****  *
560 DATA 153                : REM $99      *  **   *
```

CAUTION: Do NOT put the REMarks on line 510 thru 550, since data must
be the last statement on a line.

NOTE: The REM in line 330 is required. All other REMs are optional.

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 a filled in cell a

'1' bit, empty cells are '0'. Convert the 1's and 0's to hex notation and thence to decimal.

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

2. A more complicated program, sparsely commented.

```
100 GRAPHICS 0                 : REM not necessary, just prettier
120 PMGRAPHICS 2 : PMCLR 0 : PMCLR 1
130 SETCOLOR 2,0,0 : PMCOLOR 0,12,8 : PMCOLOR 1,12,8
140 p0 = PMADR(0) : p1 = PMADR(1) : REM addr's for 2 players
150 v0 = 60 : vold = v0          : REM starting vertical position
160 h0 = 110                     : REM starting horizontal position
200   FOR loc = v0-8 TO v0+7   : REM a 16-high double player
210   READ X
220   POKE p0+loc,INT(X/$100)
230   POKE p1+loc,X & $FF
240   NEXT loc
300 REM ANIMATE IT
310 LET radius=40 : DEG : REM 'let' required, RAD is keyword
320   WHILE 1                    : REM an infinite loop!!
330   c=int( 16 * RND(0) ) : PMCOLOR 0,c,8 : PMCOLOR 1,c,8
340    FOR angle = 0 to 355 STEP 5 : REM in degrees, remember
350    vnew = int( v0 + radius * SIN(angle) )
360    vchange = vnew – vold : REM change in vertical position
370    hnew = h0 + radius * COS(angle)
380    PMMOVE 0,hnew;vchange : PMMOVE 1,hnew+8;vchange
          : REM move two players together
390    vold = vnew
400     SOUND 0,hnew,10,12 : SOUND 1,vnew,10,12
410     NEXT angle
420   REM just did a full circle
430   ENDWHILE
500 REM the fancy DATA!            8421842184218421
510 DATA $03C0            |       ****        |
520 DATA $0C30            |     **    **      |
530 DATA $1008            |    *        *     |
540 DATA $2004            |   *          *    |
550 DATA $4002            |  *            *   |
560 DATA $4E72            |  *  ***  *** *   |
570 DATA $8A51            |* * *   * *   *|
580 DATA $8E71            |* ***   *** *|
590 DATA $8001            |*              *|
600 DATA $9009            |* *        * *|
620 DATA $4812            | * *      * * |
630 DATA $47E2            | *  ****** * |
640 DATA $2004            |  *          *    |
650 DATA $0C30            |     **    **      |
660 DATA $03C0            |       ****        |
```

Notice how much easier it is to use the hex data.

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

ERROR NUMBER	DESCRIPTION
1	While SET 0,1 was specified, the user hit the BREAK key. This TRAPpable error gives the BASIC XL programmer total system control.
2	All available memory has been used. No more statements can be entered and no more variables (arithmetic, string or array) can be defined.
3	An expression or variable evaluates to an incorrect value. Example:

An expression that can be converted to a two byte integer in the range 0 to 65235 (hex $FFFF) is called for and the given expression is either too large or negative.

A = PEEK(-1)
DIM B(70000)

Both these statements will produce a value error.

Example:

An expression that can be converted to a one byte integer in the range 0 to 255 hex(FF) is called for and the given expression is too large.

POKE 5000,750

This statement produces value error.

Example:

A=SQR(-4)   Produces a value error.

4	No more variables can be defined. The maximum number of variables is 128.
5	A character beyond the DIMensioned or current length of a string has been accessed. Example:

1000 DIM A$(3)
2000 A$(5) = "A"

This will produce a string length error at line 2000 when the program is RUN.

6	A READ statement is executed but we are already at the end of the last DATA statement.
7	A line number larger than 32767 was entered.

ERROR NUMBER	DESCRIPTION
8	The INPUT or READ statement did not receive the type of data it expected. Example:

```
1000 READ A
2000 PRINT A
3000 END
4000 DATA 12AB
```

Running this program will produce this error.

9	A previously DIMensioned string or array is DIMensioned again. Example:

```
1000 DIM A(10)
2000 DIM A(10)
```

This program produces a DIM error.

10	An expression is too complex for BASIC XL to handle. The solution is to break the calculation into two or more BASIC XL statements.
11	The floating point routines have produced a number that is either too large or too small.
12	The line number required for a GOTO or GOSUB does not exist. The GOTO may be implied as in:

```
1000 IF A=B THEN 500
```

The GOTO / GOSUB may also be part of an ON statement.

13	A NEXT was encountered but there is no information about a FOR with the same variable. Example:

```
1000 DIM A(10)
2000 REM FILL THE ARRAY
3000 FOR I = 0 TO 10
4000 A(I) = I
5000 NEXT I
6000 REM PRINT THE ARRAY
7000 FOR K = 0 TO 10
8000 PRINT A(K)
9000 NEXT I
10000 END
```

Running this program will cause the following output:

```
0
ERROR- 13 AT LINE 9000
```

ERROR NUMBER	DESCRIPTION
14	The line just entered is longer than BASIC 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	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 GOSUB or FOR is deleted, then the user types CONT and the program tries to execute the RETURN or NEXT.
Example:

```
1000 GOSUB 2000
1100 PRINT "RETURNED FROM SUB"
1200 END
2000 PRINT "GOT TO SUB"
2100 STOP
2200 RETURN
```

If this program is run the print out is:

GOT TO SUB
STOPPED   AT LINE 2100

Now if the user deletes line 1000 and then types CONT we get:

ERROR- 15 AT LINE 2200

| 16 | A RETURN was encountered but we have no information about a GOSUB. Example: |

```
1000 PRINT "THIS IS A TEST"
2000 RETURN
```

If this program is run the print out is:

THIS IS A TEST

ERROR- 16 AT LINE 2000

NOTE: Improper use of POP could also cause this error.

ERROR NUMBER	DESCRIPTION
17	If when entering a program line a syntax error occurs, the line is saved with an indication that it is in error. If the program is run without this line being corrected, execution of the line will cause this error. NOTE: The saving of a line that contains a syntax error can be useful when LISTing and ENTERing programs.
18	If when executing the VAL function, the string argument does not start with a number, this message number is generated. Example: A = VAL("ABC") Produces this error.
19	The program that the user is trying to LOAD is larger than available memory. This could happen if the user had used LOMEM to change the address at which Basic tables start, or if he is LOADing on a machine with less memory than the one on which the program was SAVEd.
20	If the device / file number given in an I/O statement is greater than 7 or less than 0, then this error is issued. Example: GET #8,A
21	This error results if the user tries to LOAD a file that was not created by SAVE.
22	This error occurs if the length of the entire format string in a PRINT USING statement is greater than 255. It also occurs if the length of the sub-format for one specific variable is greater than or equal to 60.
23	The value of a variable in a PRINT USING statement is greater than or equal to 1E+50.
24	In a PRINT USING statement, the format indicates that a variable is a numeric when in fact the variable is a string. Or the format indicates the variable is a string when it is actually a numeric. Example: PRINT USING "###",A$ PRINT USING "%%%",A Will produce this error.

ERROR NUMBER	DESCRIPTION
25	The string being retrieved by RGET from a device (i.e., the one written by RPUT) has a different DIMension length than the string variable to which it is to be assigned.
26	The record being retrieved by RGET (i.e., the one written by RPUT) is a numeric, but the variable to which it is to be assigned is a string. Or the record is a string, but the variable is a numeric.
27	An INPUT statement was executed and the USER entered CTRL-C <RETURN>.
28	The end of a control structure such as ENDIF or ENDWHILE was encountered but the run-time stack did not have the corresponding beginning structure on the Top of Stack. Example:

```
10 WHILE 1 : REM loop forever
20 GOSUB 100
100 ENDWHILE
```

ENDWHILE finds the GOSUB on Top of Stack and issues the error.

29	An illegal player/missile number. Players must be numbered from 0-3 and missiles from 4-7.
30	The user attempted to use a PMG statement other than PMGRAPHICS before executing PMGRAPHICS 1 or PMGRAPHICS 2.
32	End of ENTER. This is the error resulting from a program segment such as:

```
SET 9,1 : TRAP line# : ENTER filename
```

When the ENTER terminates normally.

34	The second aexp in a RENUM or NUM command evaluated to zero, and an increment of 0 is invalid.
35	When RENUMburing, the maximum line number (32767) was exceeded.
40	You attempted to use a string variable as a string array variable, or vice versa. Example:

```
DIM A$(3,20)
A$="THIS CAUSES AN ERROR"
```

Would create this error.

LABEL	HEXADECIMAL LOCATION	COMMENTS and DESCRIPTION
APPMHI	DE	Highest location used by BASIC
RTCLOK	12,13,14	Screen Frame Counter (1/60 sec.)
SOUNDR	41	Noisy I/O Flag (0=quiet)
ATRACT	4D	Attract Mode Flag (128=Attract)
LMRGIN, RMRGIN	52,53	Left, Right Margin (Defaults 2, 39)
RAMTOP	6A	Actual top of memory (page number)
LOMEM	80,81	BASIC XL low memory pointer
MEMTOP	90,91	BASIC XL high memory pointer (usually same as APPMHI)
FR0	D4,D5	Value returned to BASIC XL from a USR function (LSB, MSB)
MEMTOP	2E5,2E6	OS top of available memory (LSB, MSB)
MEMLO	2E7,2E8	OS low memory pointer (LSB, MSB)
CRSINH	2F0	Cursor Inhibit (0=cursor on)
CHACT	2F3	Character Mode Register (4=vertical reflect; 2=normal; 1=blank)
CHBAS	2F4	Character Set Base Register
ATACHR	2FB	Last ATASCII Character
CH	2FC	Last keyboard key pressed (keyboard matrix code)
FILDAT	2FD	Fill data for graphics Fill (XIO)
DSPFLG	2FE	Display Flag (1=display control character)
CONSOL	D01F	Console Keys (bit 2=OPTION; bit 1 SELECT; bit 0 START)
SKCTL	D20F	Serial Port Control Register (bit 2=0 if last key still pressed)

Address	Description
$00	OS Variables
$80	BASIC XL System RAM
$CB	FREE BASIC XL RAM
$D2	Atari Floating Point Registers
$100	Hardware Stack
$200	OS Variables
$3C0	Printer Buffer
$3E8	OS RAM
$3FD	Cassette Buffer
$480	BASIC XL Stack and Miscellaneous Variable
$57E	Input and Floating Point Buffers
$680	Free RAM
$700	DOS RAM
(MEMLO)	BASIC XL program, buffers, tables, run-time stack.
(APPMHI)	Free RAM
(MEMTOP)	Screen Memory also optional P/M memory
$A000	BASIC XL Cartridge
$C000	O.S., ROMs, etc.
$D000	Hardware Registers
$D800	OS and Floating Point ROM
$FFFF	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
0	0	♥	18	12	▬	36	24	$
1	1	├	19	13	┼	37	25	%
2	2	│	20	14	●	38	26	&
3	3	┘	21	15	▬	39	27	'
4	4	┤	22	16	│	40	28	(
5	5	┐	23	17	┬	41	29	)
6	6	╲	24	18	┴	42	2A	✳
7	7	╱	25	19	▌	43	2B	+
8	8	◢	26	1A	└	44	2C	,
9	9	▪	27	1B	␛	45	2D	–
10	A	◣	28	1C	✚	46	2E	.
11	B	▪	29	1D	✚	47	2F	/
12	C	■	30	1E	←	48	30	0
13	D	▬	31	1F	→	49	31	1
14	E	▬	32	20	Space	50	32	2
15	F	▪	33	21	!	51	33	3
16	10	✙	34	22	"	52	34	4
17	11	▛	35	23	#	53	35	5

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
54	36	6	75	4B	K	96	60	♣
55	37	7	76	4C	L	97	61	a
56	38	8	77	4D	M	98	62	b
57	39	9	78	4E	N	99	63	c
58	3A	:	79	4F	O	100	64	d
59	3B	;	80	50	P	101	65	e
60	3C	<	81	51	Q	102	66	f
61	3D	=	82	52	R	103	67	g
62	3E	>	83	53	S	104	68	h
63	3F	?	84	54	T	105	69	i
64	40	@	85	55	U	106	6A	j
65	41	A	86	56	V	107	6B	k
66	42	B	87	57	W	108	6C	l
67	43	C	88	58	X	109	6D	m
68	44	D	89	59	Y	110	6E	n
69	45	E	90	5A	Z	111	6F	o
70	46	F	91	5B	[	112	70	p
71	47	G	92	5C	\	113	71	q
72	48	H	93	5D	]	114	72	r
73	49	I	94	5E	^	115	73	s
74	4A	J	95	5F	_	116	74	t

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
117	75	U	137	89		157	9E	
118	76	V	138	8A		158	9E	
119	77	W	139	8B		159	9F	
120	78	X	140	8C		160	A0	
121	79	Y	141	8D		161	A1	
122	7A	Z	142	8E		162	A2	
123	7B		143	8F		163	A3	
124	7C		144	90		164	A4	
125	7D		145	91		165	A5	
126	7E		146	92		166	A6	
127	7F		147	93		167	A7	
128	80		148	94		168	A8	
129	81		149	95		169	A9	
130	82		150	96		170	AA	
131	83		151	97		171	AB	
132	84		152	98		172	AC	
133	85		153	99		173	AD	
134	86		154	9A		174	AE	
135	87		155	9B	EOL	175	AF	
136	88		156	9C		176	B0	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
177	B1	1	197	C5	E	217	D9	Y
178	B2	2	198	C6	F	218	DA	Z
179	B3	3	199	C7	G	219	DB	[
180	B4	4	200	C8	H	220	DC	\
181	B5	5	201	C9	I	221	DD	]
182	B6	6	202	CA	J	222	DE	^
183	B7	7	203	CB	K	223	DF	_
184	B8	8	204	CC	L	224	E0	`
185	B9	9	205	CD	M	225	E1	a
186	BA	:	206	CE	N	226	E2	b
187	BB	;	207	CF	O	227	E3	c
188	BC	<	208	D0	P	228	E4	d
189	BD	=	209	D1	Q	229	E5	e
190	BE	>	210	D2	R	230	E6	f
191	BF	?	211	D3	S	231	E7	g
192	C0	@	212	D4	T	232	E8	h
193	C1	A	213	D5	U	233	E9	i
194	C2	B	214	D6	V	234	EA	j
195	C3	C	215	D7	W	235	EB	k
196	C4	D	216	D8	X	236	EC	l

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
237	ED		244	F4		251	FB	
238	EE		245	F5		252	FC	
239	EF		246	F6		253	FD	
240	F0		247	F7		254	FE	
241	F1		248	F8		255	FF	
242	F2		249	F9				
243	F3		250	FA				

NOTE:

1. ATASCII stands for "ATARI ASCII". Letters and numbers have the same values as those in ASCII, but some of the special characters are different.
2. Characters from 128-255 are reverse colors of 1 to 127.
3. Add 32 to upper case code to get lower case code for same letter.
4. To get ATASCII code, tell computer (direct mode) to PRINT ASC ("_____"). Fill blank with letter, character, or number of code. Must use the quotes.

All keywords, grouped by statements and then functions, are listed below in alphabetical order. A page number reference is given to enable the user to quickly find more information about each keyword.

EXPLANATION OF TERMS

exp	-	EXPression	line	Line number (can be aexp)
aexp	-	Arithmetic EXP	pm	Player/Missile number (aexp)
sexp	-	String EXP	[xxx]	xxx is optional
var	-	VARiable	[xxx…]	xxx is optional, and may be repeated
avar	-	Arithmetic VAR	<stmts>	One or more statements
svar	-	String VAR	addr	ADDRess aexp, must be 0 – 65535
mvar	-	Matrix VAR (or element)		
fn	-	File Number		
filename	-	svar or string literal (quotes are optional except with LIST)		

NOTE: Keywords denoted by an asterisk (*) not in Atari BASIC.

Generally, BASIC XL is totally compatible with Atari BASIC. Virtually all programs written in Atari BASIC and SAVEd or CSAVEd will LOAD or CLOAD properly with BASIC XL and run without changes. However, in a few very subtle ways, there are minor differences between Atari BASIC and BASIC XL. This appendix presents a list of known differences, but OSS cannot guarantee that it is an exhaustive list.

1. Variable Names

When programs are SAVEd or CSAVEd under Atari BASIC and then LOADed or CLOADed under BASIC XL, there will never be a conflict in variable name usage. However, when a program is LISTed from Atari BASIC and then ENTERed into BASIC XL, or when a program listing published in a magazine or book is typed into BASIC XL, it is possible that BASIC XL will not accept lines of code which are valid in Atari BASIC.

The reason, of course, is that BASIC XL has a much richer range of keywords for statements and functions than does Atari BASIC, and in neither language can a variable name begin with a statement name unless it is preceded with a LET keyword. To illustrate the problem, let us examine the following valid Atari BASIC line:

NUMBER = 7

Because NUM is a valid BASIC XL statement name, it will now be seen by our syntax parsers as this:

NUM BER = 7

That is, it is seen as a NUM command with a starting line number of (BER=7). Since you probably don't have a variable named BER in your program, BER will not equal 7, so the statement becomes the equivalent of simply

NUM 0

which is certainly not what was intended.

In most cases, variable name conflicts such as this will result in a syntax error. In this particular case (and a few others), the result appears valid to BASIC XL so no syntax error results. How can you detect such problems easily? The easiest way is to examine the LISTed form of the program. Since BASIC XL always lists a space after every keyword, and since all keywords and variables

are listed in lower case except for the first letter, it is often easy to spot discrepancies of this form.

In any case, the intent of the original Atari BASIC program can always be accomplished by simply placing the LET keyword in front of the offending variable. Thusly:

LET NUMBER = 7

In the case of array variables, the situation is both simpler and more complex. Only those variables which have EXACTLY the same name is a new BASIC XL function (such as BUMP or RANDOM) will be in conflict, so the number of offending names is much smaller. However, the only fix that can be made in these cases is to change the name of the variable, usually by simply adding a single character (e.g., change BUMP to BUMPS).

2. Upper and Lower Case, Inverse Video

Again, these problems will never occur with programs SAVEd in Atari BASIC and LOADed under BASIC XL.

In order to make keyboard entry more flexible and more consistent, BASIC XL allows you, the programmer, to type your programs in with upper case letters, lower case letters, or even inverse video characters. BASIC XL accomplishes this by simply changing all such characters to their conventional normal video, upper case counterparts, excepting ONLY those characters enclosed in quote marks.

The only times that this makes any difference at all are (1) when the user types in a string and does not terminate it with a quote mark and (2) in DATA and REM statements where the user really desired the lower case or inverse characters. In either case, enclosing the desired characters in matching quotes will solve the problem (recall that BASIC XL supports quoted strings in DATA statements).

However, BASIC XL also provides a means of completely emulating Atari BASIC in this regard, should you wish. Simply use the command

SET 5,0

and all characters will remain unconverted. this is also handy when ENTERing programs LISTed from Atari BASIC.

This same SET has a secondary effect: when non-converting, upper case only entry is selected, then all LISTings will be in upper case only. This allows the BASIC XL user to LIST

programs which will be compatible with Atari BASIC's ENTER capability (providing, of course, that no advanced statements or functions were used in the code).

3. Program Which RUN Too Fast

Of couse, the fact that your programs will run faster is probably one of the primary reasons that you bought BASIC XL. And, generally, the speed-up provided is only beneficial.

A few programs, though, will depend on timing loops, etc., to run properly. There is no real "cure" for this "problem". Hopefully, you will be able to play the faster games and/or read the faster messages.

A related problem has to do with the fact that BASIC XL always automatically executes a FAST command whenever it encounters a statement of the form

            RUN filename

(That is, ONLY when a filename is given in conjunction with RUN).

Many programs which run only somewhat faster with normal BASIC XL will run much, much faster when the FAST command is given. You may really find yourself with a game which is simply too fast to play.

There are two solutions. The first is simply to LOAD the program first and then issue a separate RUN command. If, however, you have an auto-booting disk or a program which chains to another program via RUN, this is not a practical solution. The second solution, then, is to simply hold down the SELECT button when the RUN is executed (which may imply holding the button for a while when an auto-booting disk is started). BASIC XL allows this usage of SELECT as a means of telling it to slow down.

BASIC XL attempts to conforms to all memory location usage published in any or all of the following books:

Atari BASIC Reference Manual, by Atari, Inc.

Operating System Source Listing, for Atari 400/800, Atari, Inc. (Except that locations SIN, COS, ATAN, and SQR are incorrect, even for Atari BASIC).

De Re Atari, by Chris Crawford, et al

Mapping the Atari, from COMPUTE! Books

Master Memory Map, by Educational Software, Inc.

A few programs written by extremely knowledgeable individuals have, in the past, 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 memory locations have special meanings to Atari BASIC.
3) Certain subroutines, internal to Atari BASIC, are located at certain addresses.

Obviously, it was impossible to add the features and speed to BASIC XL which we did without adding code and making more use of the memory reserved for BASIC. Although we attempted to keep the changes to an absolute minimum, we cannot possibly be responsible for maintaining compatibility with programs which use such undocumented and unpublished information.

May we remind you of the memory locations and map which we presented in Appendices B and C. We invite comparison of these with Appendices D and I in the Atari BASIC Reference Manual. All usage is compatible.

Finally, for those who are experienced programmers, we present here a list of all zero page locations which ARE used in the same way by both Atari BASIC and BASIC XL. Only addresses are given. Refer to a memory map book or The Atari BASIC Sourcebook (published by COMPUTE! Books) for descriptions of the locations' uses.

$80 to $92	$94 to $B3
$B6 to $B8	$BA to $BB
$C2 to $C3	$C8 to $C9
$D2 to $FF	

CAUTION: Some of these locations may be used by BASIC XL for additional purposes, beyond (but compatible with) the usages of Atari BASIC. These additional purposes may imply use of the locations at times when they were unused by Atari BASIC or even use of certain bits left unmodified by Atari BASIC. It is suggested that the user should not modify these locations, though he might profitably use the information they contain. Additionally, OSS reserves the right to change usage of these locations if necessary for future corrections or improvements, though you may safely assume that those locations mentioned in "Mapping the Atari" will remain unchanged.

5. Automatic String Dimensions

BASIC XL automatically dimensions strings to 40 characters. Again, this should have no affect on currently running Atari BASIC programs. If desired, you can use

SET 11,0

to ensure compatibility.

6. INDENTED LISTINGS

When BASIC XL lists a program, it automatically adds indentation for FOR…NEXT loops (and other control structures). This could only be a problem with long lines LISTed to disk and when re-ENTERed into BASIC. Again, you may use

SET 12,0

to ensure compatibility and remove the indenting.

A

B

C

D

E

END, 20, 30, 44, 45, 47, 48, 50, 52, 59, 60, 61, 67, 79, 81, 104, 119, 120, 134, 135

ENTER, 17, 20, 34, 42, 43, 54, 55, 64, 69, 71, 72, 137, 148

ERASE, 64, 70

ERR, 34, 62, 87, 88, 104

ERROR DESCRIPTIONS, 133

Execute Mode, 18

exp, 15, 16, 17, 29, 72, 74, 75, 84, 145

EXP, 88, 89, 90, 145

Expression, 16, 17, 18, 30

Expressions, 17, 29, 30, 31

F

FAST, 14, 33, 35, 40, 148

filespec, 17, 36, 37, 40, 65, 66, 69, 70, 72, 73, 80, 82, 85, 86, 118

FIND, 88, 94, 98

FOR, 20, 35, 41, 42, 43, 44, 45, 48, 50, 60, 61, 81, 92, 93, 98, 101, 120, 127, 130, 131, 134, 135, 150

FOR (F.)…TO…STEP / NEXT, 48

FRE, 17, 88, 104

Function, 17

G

Game Controller Functions, 88, 99

GET, 64, 70, 80, 86, 87, 114, 115, 126, 136

GOSUB
GOSUB, 16, 35, 38, 39, 44, 45, 50, 52, 59, 60, 134, 135, 137

GOTO, 16, 35, 39, 40, 42, 43, 44, 45, 50, 52, 53, 54, 57, 59, 60, 63, 87, 90, 94, 98, 101, 104, 108, 118, 120, 130, 134

GRAPHICS, 47, 54, 90, 92, 101, 108, 109, 110, 111, 112, 113, 114, 117, 118, 121, 123, 124, 125, 127, 131

H

HEX$, 88, 104

HSTICK, 88, 99, 100, 101

I

IF / THEN, 52, 54, 59

IF…ELSE…ENDIF, 45, 54, 56

INPUT, 31, 41, 42, 54, 60, 64, 71, 72, 73, 86, 90, 94, 98, 104, 108, 115, 134, 137

INT, 17, 27, 30, 88, 89, 90, 92, 131

K

Keyword, 15, 17

L

LEFT$, 88, 95, 98

LEN, 31, 46, 67, 84, 88, 95, 98

LET, 20, 45, 57, 130, 131, 146, 147

lineno, 15, 17, 35, 36, 39, 44, 50, 52, 53, 54, 59, 62

BASIC XL
The Right Tool For
All Atari® BASIC Programming

Programs Run Faster: A special FAST command precompiles the program currently in memory. Programs run 2 to 4 times faster than they would using Atari BASIC.

Write Better Programs Faster: BASIC XL will automatically number program liines and renumber entire programs on request. The LIST command displays your program in an easy-to-read format.

More Built-in Features: Offers many advanced commands and special functions that up to now were only available on other computer systems...string arrays and search...disk commands directly from BASIC...extended input and output...English like error messages, and program debugging with the TRACE command. And best of all BASIC XL still uses only 8K of RAM!

Simplifies Using Atari Graphics: Ten NEW graphic commands offer you uncomplicated, direct access to using player/missiles, animation, and dazzling computer graphics.