For 8-Bit ATARI®

BASIC XL

Cartridge System, Run Time & Tool Kit

Compatible !

Flexible !

Fast !

Precision
Software Tools

A Division of ICD, Incorporated

A Trademark of FTe
Revised by GoodByteXL

Last Edit: 31 January 2Ø22

Preface

As tribute to OSS this fully edited version is provided to keep one of the best and fastest BASIC versions on ATARI 8-bit computers accessible to all being interested in this great piece of software.

BASIC XL was released by OSS in 1983.
OSS merged into a division of ICD, Inc. in 1988.
ICD sold their ATARI 8-bit product line to FTe in 1993.
FTe disappeared supposedly in 1996.
The author Bill Wilkinson passed away in November 2015.
The current status of BASIC XL is that the Wilkinson Family gave allowance to atariwiki.org to use OSS material.

Many of those famous A8 hardware and software products from OSS and ICD are still available thanks to a worldwide active A8 community. So is this manual.

My BASIC XL reference manual went through scanner and OCR process as the one about Action!.

The new BASIC XL manual in chapters 1 to 8 keeps the original layout for reference to the original paperback and of course for historical reasons. Nearly all entries from the original's table of contents are found on the same page.

The BASIC XL Toolkit including The BASIC XL Runtime Package was added to this manual as chapter 9 and is adapted to the layout as deemed necessary.

Therefore the appendices needed amendment and changes as well.

Typos and errors found were corrected and information concerning the XL/XE computers were added to the new BASIC XL manual.

BASIC XL and other OSS programming languages work best together with SpartaDOS X V. 4.47 or newer provided by DLT.

Enjoy BASIC XL ... and may your A8 always be with you!

GoodByteXL, January 2022

P.S.: Special thanks for help to Roland.

TABLE OF CONTENTS

Chapter 1: Introduction
1.1 Features of BASIC XL 1
1.2 Special Notations used in this Manual 2
1.3 Glossary and terminology 3
1.4 Operating Modes 7

Chapter 2: VARIABLES, OPERATORS, EXPRESSIONS
2.1 Variables (var) 9
2.1.1 Arithmetic variables (avar) 1Ø
2.1.2 Array / Matrix Variables (mvar) 1Ø
2.1.3 String Variables (svar) 12
2.1.4 String Array Variables (svar) 12
2.1.5 DIM 13
2.2 Operators 14
2.2.1 Arithmetic Operators (aop) 14
2.2.2 Logical Operators (lop) 15
2.2.3 Operator Precedence 16
2.3 Expressions (exp) 17
2.3.1 Numbers 17
2.3.2 Arithmetic Expressions (aexp) 18
2.3.3 String Expressions (sexp) 19

Chapter 3: PROGRAM DEVELOPMENT COMMANDS
3.1 BYE (B.) 21
3.2 CLR 21
3.3 CONT (CON.) 22
3.4 DEL 22
3.5 DOS 23
3.6 FAST 23
3.7 LIST (L.) 24
3.8 LOMEM 24
3.9 LVAR (LV.) 25
3.1Ø NEW 25
3.11 NUM 25
3.12 REM (R.) 26
3.13 RENUM 27
3.14 RUN 27
3.15 SET 28
3.16 STOP 31
3.17 TRACE and TRACEOFF 31

Chapter 4: PROGRAM CONTROL STATEMENTS
4.1 Assignment Statement 33
4.2 END 34
4.3 FOR(F.)...TO...STEP / NEXT(N.) 35
4.4 GOSUB (GOS.) / RETURN (RET.) 36
4.5 GOTO (G.) 37
4.6 IF/THEN 39
4.7 IF...ELSE...ENDIF 4Ø
4.8 LET 41
4.9 MOVE 42
4.1Ø ON... 43
4.11 POP 44
4.12 RESTORE (RES.) 45

TABLE OF CONTENTS

4.13 TRAP (T.) 45
4.14 WHILE...ENDWHILE 46

Chapter 5: INPUT/OUTPUT COMMANDS AND DEVICES
5.1 Comments and Notations 47
5.2 BGET 49
5.3 BPUT 5Ø
5.4 CLOAD 5Ø
5.5 CLOSE (CL.) 5Ø
5.6 CSAVE (CS.) 51
5.7 DATA (D.) 51
5.8 DIR 52
5.9 ENTER (E.) 52
5.1Ø ERASE 53
5.11 GET 53
5.12 INPUT (I.) 53
5.12.1 Advanced use of INPUT 54
5.13 LOAD (LO.) 55
5.14 LPRINT (LP.) 55
5.15 NOTE (NO.) 55
5.16 OPEN (O.) 56
5.17 POINT (P.) 57
5.18 PRINT (PR or ?) 57
5.19 PRINT USING 58
5.2Ø PROTECT 63
5.21 PUT (PU.) 63
5.22 READ 63
5.23 RENAME 64
5.24 RGET 64
5.25 RPUT 65
5.26 SAVE (S.) 66
5.27 STATUS (ST.) 66
5.28 TAB 66
5.29 UNPROTECT (UNP.) 67
5.3Ø XIO (X.) 67
5.31 An Example Program 68

Chapter 6: FUNCTION LIBRARY
6.1 Arithmetic Functions 69
6.1.1 ABS 69
6.1.2 CLOG 69
6.1.3 EXP 7Ø
6.1.4 INT 7Ø
6.1.5 LOG 7Ø
6.1.6 RANDOM 7Ø
6.1.7 RND 71
6.1.8 SGN 71
6.1.9 SQR 71
6.1.1Ø An Example Program 71
6.2 Trigonometric Functions 72
6.2.1 ATN 72
6.2.2 COS 72
6.2.3 DEG and RAD 72
6.2.4 SIN 72

6.2.5 An Example Program 73
6.3 String Functions 73
6.3.1 ASC 73
6.3.2 CHR$ 73
6.3.3 FIND 74
6.3.4 LEFT$ 75
6.3.5 LEN 75
6.3.6 MID$ 75
6.3.7 RIGHT$ 76
6.3.8 STR$ 76
6.3.9 VAL 76
6.3.1Ø An Example Program 77
6.4 Game Controller Functions 78
6.4.1 HSTICK 78
6.4.2 PADDLE 78
6.4.3 PEN 78
6.4.4 PTRIG 78
6.4.5 STICK 79
6.4.6 STRIG 79
6.4.7 VSTICK 79
6.4.8 An Example Program 8Ø
6.5 Player/Missile Functions 8Ø
6.5.1 BUMP 8Ø
6.5.2 PMADR 81
6.6 Special Purpose Functions 81
6.6.1 ADR 81
6.6.2 DPEEK 81
6.6.3 DPOKE 82
6.6.4 ERR 82
6.6.5 FRE 82
6.6.6 HEX$ 83
6.6.7 PEEK 83
6.6.8 POKE 83
6.6.9 SYS 84
6.6.1Ø TAB 84
6.6.11 USR 84
6.6.12 An Example Program 86

Chapter 7: SCREEN GRAPHICS AND SOUND
7.1 GRAPHICS (GR.) 87
7.1.1 GRAPHICS Mode Ø 88
7.1.2 GRAPHICS Modes 1 and 2 88
7.1.3 GRAPHICS Modes 3, 5 and 7 89
7.1.4 GRAPHICS modes 4,6 89
7.1.5 GRAPHICS mode 8 .... 9Ø
7.1.6 GRAPHICS modes 9, 1Ø, and 11 .... 9Ø
7.1.7 GRAPHIC modes 12 and 13 91
7.1.8 GRAPHIC modes 14 and 15 91
7.2 COLOR (C.) 92
7.3 DRAWTO (DR.) 93
7.4 LOCATE (LOC.) 93
7.5 PLOT (PL.) 94
7.6 POSITION (POS.) 94
7.7 PUT and GET (as applied to graphics) 94

TABLE OF CONTENTS

7.8 SETCOLOR (SE.) 95
7.9 XIO (X.) Special Fill Application 97
7.1Ø SOUND (SO.) 98

Chapter 8: PLAYER / MISSILE GRAPHICS
8.1 An Overview of P/M Graphics 1Ø1
8.2 P/M Graphics Conventions 1Ø3
8.3 BGET and BPUT with P/M's 1Ø3
8.4 PMCLR 1Ø4
8.5 PMCOLOR (PMCO.) 1Ø4
8.6 PMGRAPHICS (PMG.) 1Ø4
8.7 PMMOVE 1Ø6
8.8 PMWIDTH (PMW.) 1Ø7
8.9 POKE and PEEK with P/M's 1Ø7
8.1Ø MISSILE (MIS.) 1Ø7
8.11 MOVE with P/M's 1Ø8
8.12 USR with P/M's 1Ø8
8.13 Example PMG Programs 1Ø9

Chapter 9: THE BASIC XL TOOLKIT
9.1 THE BASIC XL RUNTIME PACKAGE 114
9.1.1 How Does the RUNTIME Package Work? 114
9.1.2 How Do You Use the RUNTIME Package? 114
9.1.3 Statements that can NOT be used with RUNTIME .. 115
9.1.4 Error Handling In RUNTIME BASIC XL 116
9.1.5 RunTime Restart 116
9.1.6 Incompatibilities 116
9.2 BASIC XL Example Programs 116
9.2.1 MENU.BXL 119
9.2.2 SNAILS 123
9.2.3 PICOADV 127
9.2.4 LEM 135
9.2.5 GTIATEST 14Ø
9.2.6 CIRCLES 142
9.2.7 DISKIO 143
9.2.7.1 SIO and the Device Control Block 144
9.2.7.2 The Sector Access Routine 145
9.2.7.3 Technical Sidelight 146
9.2.8 CONFIG 147
9.2.8.1 The Percom Standard 148
9.2.8.2 Reading and Writing the Config Block 149
9.2.9 PHONE 153
9.2.9.1 Sequential and Other Files 153
9.2.9.2 How to Use NOTE and POINT to Advantage 154
9.2.9.3 The Concept Behind PHONE.BXL. alias BlackBook ... 156
9.2.9.4 BlackBook Data Files 156
9.2.9.5 BlackBook Index Files 157
9.2.9.6 The Index String 158
9.2.9.7 Program Description: PHONE.BXL, BlackBook ... 159
9.2.1Ø MAKEAUTO 167
9.3 BASIC XL Extended Statements 168
9.3.1 How to Install the Extended Statements 168
9.3.2 Abbreviations Used In Formal

Statement Definitions ... 170
9.3.3 Procedure Blocks and Related Statements 171
9.3.3.1 PROCEDURE (PROC.) 174
9.3.3.1.1 Secondary Considerations 177
9.3.3.2 CALL 180
9.3.3.2.1 Secondary Considerations 181
9.3.3.3 LOCAL 182
9.3.3.3.1 Secondary Considerations 183
9.3.3.4 EXIT 184
9.3.3.4.1 Secondary Considerations 185
9.3.4 Sorting String Arrays 187
9.3.4.1 SORTUP 191
9.3.4.2 SORTDOWN 192
9.4 Example BASIC XL Programs with Extended Statements ... 194
9.4.1 FACTOR.BXE 194
9.4.2 SORTDIR.BXE 196
9.4.3 SORTNUM.BXE 197
9.4.4 GTIATEST.BXE 200
9.4.5 DISKIO.BXE 200
9.4.6 PHONE.BXE 201

Appendices
Appendix A: ERROR DESCRIPTIONS 203
Appendix B: SYSTEM MEMORY LOCATIONS 209
Appendix C: BASIC XL MEMORY MAP 211
Appendix D: ATASCII CHARACTER SET 213
Appendix E: SYNTAX SUMMARY AND KEYWORD INDEX 217
Appendix F: COMPATIBILITY WITH ATARI BASIC 221
Appendix G: BENCHMARKS 227

Chapter 1: Introduction

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 gives 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 1/2 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
1ØØ	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.

Items in Brackets

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.

Items Stacked Vertically in Bars

Items stacked vertically in bars indicate that anyone 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 |
```

Command abbreviations in headings

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) Generally composed of a variable,' function, 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 Ø 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 12Ø 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 = 1ØØ, X is a variable and 1ØØ 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 5.1.

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 6 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 mode BASIC program. Must be an integer from Ø 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 or 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 4ØØ and 8ØØ.

2.1 Variables (var)

There are two basic types of variables in BASIC XL -- arithmetic variables and string variables. Also, there are 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 than 128 characters long. All string or string array variable names must end in the '$' (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 one use
200 FOR 1=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 locations (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 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
     +-----+-----+-----+-----+-----+
     | Ø,Ø | Ø,1 | Ø,2 | Ø,3 | Ø,4 |
     +-----+-----+-----+-----+-----+
   R | 1,Ø | 1,1 | 1,2 | 1,3 | 1,4 |
     +-----+-----+-----+-----+-----+
   O | 2,Ø | 2,1 | 2,2 | 2,3 | 2,4 |
     +-----+-----+-----+-----+-----+
   W | 3,Ø | 3,1 | 3,2 | 3,3 | 3,4 |
     +-----+-----+-----+-----+-----+
     | 4,Ø | 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 (Ø,Ø), 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)<1ØØ THEN...
IF ABS(ARRAY(7))<1ØØ THEN...
IF ABS(MATRIX(4,3))<1ØØ THEN...
```

2.1.3 String Variables (svar)

String variables are used to store literal strings of characters. A literal string of characters is simply a group 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 zeroth, but with strings the first element is the first, e.g.:

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

A$(1)="A", and A$(0) generates an error because the first element of a string is (1), not (0) (as in arrays and matrices).

2.1.4 String Array Variables (savar)

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$(4,4Ø)
DIM A$(1Ø,1ØØ)
```

The first example dimensions a string array called "Strarray$" with 4 elements. Each element is a string 4Ø characters long. The second example dimensions the string array "A$" to 1Ø elements, with each element being 1ØØ characters in length.

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

```
1ØØ DIM A$(3,6)
2ØØ A$(l:)-"TEST"
3ØØ A$(2: )="STRING"
4ØØ A$(3: )="ARRAY"
```

2.1.5 DIM

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

 Example:  DIM A(1ØØ)
           DIM M(6,3)
           DIM B$(2Ø)
           DIM A$(2Ø,4Ø)
```

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 l~l 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 2Ø 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 2Ø elements, with each string element being 4Ø 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 8 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)
```

% returns a value dependent on this table:

Bit A	Bit B	Bit-wise Or Result
1	1	1
Ø	1	1
Ø	Ø	Ø
1	Ø	1

```
Example: 5 % 39 -- ØØØØØ1Ø1  (5)
                   ØØ1ØØ111  (39)
                % ----------
                   ØØ1ØØ111  (result of % is 39)
```

! returns a value dependent on this table:

Bit A	Bit B	Bit-wise XOR Result
1	1	Ø
1	Ø	1
Ø	Ø	Ø
Ø	1	1

```
Example: 5 ! 39 -- ØØØØØ1Ø1  (5)
                   ØØ1ØØ111  (39)
                ! ----------
                   ØØ1ØØØ1Ø  (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 greater 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 Ø results if the test proved false.

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

```
AND -- Logical AND
OR  -- Logical OR
```

Examples:

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

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 or 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 Operations
< > = <= >= <>	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:
4Ø27	-67.254	4.33E2
-2	325.Ø4	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 1Ø squared").

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

```
$4A3Ø      -$ØA      $6FF
 -$E      -$AØØØ     $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 1Ø BCD digits and a one byte exponent.

The most significant bit of the exponent byte gives the sign of the mantissa (Ø 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 1ØØ (not powers of 1Ø).

Example:

```
        Ø.Ø2 = 2 * 1Ø^-2 = 2 * 1ØØ^-1

exponent     = -1 + 4Ø = 3F

   Ø.Ø2      = 3F Ø2 ØØ ØØ ØØ ØØ
```

The implied decimal point is always to the right of the first byte. An exponent less than hex 4Ø indicates a number less than 1. An exponent greater than or equal to hex 4Ø 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 1Ø significant digits in the special case where there are an even number of digits to the right of the decimal point (Ø,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)	/,*,+,*	3Ø
"ABC">"DEF"+7*(ASC("A"))	>,ASC,*,+	455
X=1ØØ : 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 is eth element of SA$	same as S$, except string is eth element of SA$
SA$(e;n)	same as S$(n), except string is eth element of SA$	same as S$(n), except string is eth element of SA$

String	Definition when Destination String	Definition when Source String
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 implicitly) 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
CLR	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 stored 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 <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.

Example:

```
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]]]
```

Examples:

```
LIST
LIST 1Ø
LIST 1Ø,1ØØ
LIST 1Ø,
LIST "P:"
LIST "D:DEMO.LST"
LIST "P:",2Ø,1ØØ
```

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)+1Ø24
```

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.

3.1Ø 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 5Ø
NUM ,1
NUM 5Ø,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 1Ø. If there

is no current program, NUM will start with line number 1Ø.

If the starting line number alone is given (second example), NUM will start numbering from that line number in increments of 1Ø.

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

Examples:
```
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]

Examples: 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 expressions (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. Example:

```
ON X GOSUB 100,3*Y,200
```

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

3.14 RUN

```
Format:   RUN [filespec]

Examples: 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.

Examples:
```
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 exercise 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 "?").

aexp1		aexp2	Meaning
3	(Ø)	Ø	- FOR...NEXT loops always execute at least once (ala ATARI BASIC).
		1	- FOR loops may execute zero times (ANSI standard)
4	(Ø)	Ø	- On a multiple variable INPUT, if the user enters too few items, he is reprompted (e.g., with "??")
		1	- Instead of reprompting, a TRAPable error occurs.
5	(1)	Ø	- 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 letters are converted to upper case and inverse video characters are uninverted. 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,Ø.
6	(Ø)	Ø	- Print error messages along with error numbers (for most errors)
		1	- Print only error numbers.
7	(Ø)	Ø	- 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 visa versa.
8	(Ø)	Ø	- Don't push (PHA) the number of parameters to a USR call on the stack [advantage: some assembly language· subroutines not expecting parameters may be called by a simple USR(addr)].
		1	- DO push. the count of parameters (ATARI BASIC standard).

aexp1 aexp2 Meaning

```
-----     -----    -------
  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. To move this
                       player, you need only do a PMMOVE
                       of one of the missiles since they
                       are all grouped together.

 11   (40)    1      - BASIC XL will DIM a string to this
            thru       size if you do not use a DIM
            255        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
                       statements (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

Formats: TRACE
TRACEOFF

Examples: 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$="CONCATENATED"
          C$=S$," WHICH IS ",M$
```

The 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 one 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$(LEN(A$)+1)=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

        Examples: FOR X = 1 TO 1Ø
                  NEXT X

                  FOR Y = 1Ø TO 2Ø STEP 2
                  NEXT Y

                  FOR INDEX = Z TO 1ØØ * 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.

```
1Ø FOR X=1 TO 3          : REM START OF OUTER LOOP
2Ø PRINT "OUTER LOOP"
3Ø Z=Ø
4Ø Z=Z+2
5Ø FOR Y=1 TO 5 STEP Z   : REM START OF INNER LOOP
6Ø PRINT" INNER LOOP"
7Ø NEXT Y                : REM END OF INNER LOOP
8Ø NEXT X                : REM END OF OUTER LOOP
9Ø 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. Tn 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 Ø, 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
400         360
200         150
```

4.5 GOTO (G.)

Format: [lineno) GOTO aexp

Examples: 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 7Ø, 8Ø, and 9Ø are ignored temporarily while the program executes the GOTO 1ØØ 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 7Ø. (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...]

        Examples: 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 statement is true and/or non-zero, the program executes 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<>l OR AFTER
   'YES', IT IS REALLY 1' IS DISPLAYED.

   ENTER A ?                                    (entered 1)

   A=l
   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

        Examples: 2ØØ IF A>1ØØ:PRINT "TOO BIG"
                  21Ø A=1ØØ
                  22Ø ELSE:PRINT "A-OK"
                  23Ø ENDIF

                 1ØØØ 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 LETTER$="a"
          LET AND$="*",AS,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 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 $Ø999, $8999, $499

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.

4.1Ø ON...

```
Format:   ON aexp |GOTO | lineno [,lineno...]
                  |GOSUB|

EXAMPLES: 1ØØ ON X GOTO 2ØØ,3ØØ,4ØØ
          1ØØ ON A GOSUB 1ØØØ,2ØØØ
          1ØØ ON SQR(X) GOTO 3Ø,1Ø,1ØØ
```

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 Ø 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:

```
1Ø X-X+1
2Ø ON X GOTO 1ØØ,2ØØ,3ØØ,4ØØ,5ØØ
3Ø IF X>5 THEN PRINT "COMPLETE.":END
4Ø GOTO 1Ø
5Ø END
1ØØ PRINT "NOW WORKING AT LINE 1ØØ":GOTO 1Ø
2ØØ PRINT "NOW WORKING AT LINE 2ØØ":GOTO 1Ø
3ØØ PRINT "NOW WORKING AT LINE 3ØØ":GOTO 1Ø
4ØØ PRINT "NOW WORKING AT LINE 4ØØ":GOTO 1Ø
5ØØ PRINT "NOW WORKING AT LINE 5ØØ":GOTO 1Ø
```

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

```
NOW WORKING AT LINE 1ØØ
NOW WORKING AT LINE 2ØØ
NOW WORKING AT LINE 3ØØ
NOW WORKING AT LINE 4ØØ
NOW WORKING AT LINE 5ØØ
COMPLETE
```

4.11 POP

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: 1ØØ RESTORE
                 22Ø 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:

```
   1Ø FOR N=2 TO 1 STEP -1
   2Ø RESTORE 8Ø+N
   3Ø READ A,B
   4Ø M=A+B
   5Ø PRINT "TOTAL EQUALS ";M
   6Ø NEXT N
   7Ø END
   81 DATA 3Ø,15
   82 DATA 1Ø,2Ø
```

On the first pass through the loop, A will be 1Ø and B will be 2Ø so the total in line 5Ø will print: TOTAL EQUALS 3Ø, but on the second pass, A will equal 3Ø and B will equal 15, so the PRINT statement in line 58 will display: TOTAL EQUALS 45.

4.13 TRAP (T.)

```
        Format:  TRAP aexp

        Example: 1ØØ TRAP 12Ø
```

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 find 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 : <statements> 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.

```
        Examples: WHILE 1 : ...
                  The loop executes forever

                  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 your DOS manual for more information).

NOTE: if you use D: without a drive number, D1: is assumed.
NOTE by GBXL: If the DOS used can handle current drives like e.g. SpartaDOS, it will refer to the current drive instead.

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 Ø, 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 85Ø Module -- handles Output only. Usually P: is used for a parallel printer, so it has come to mean 'Printer' as well as 'Parallel Port'.

NOTE by GBXL: P: also refers to SIO direct connect printers. Atari printers can use P1 to P4; see manual.

R1: - R4: The four RS-232 Serial Ports on the Atari 85Ø 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.

NOTE: if you use R: without a device number, R1: is assumed.

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

Ø-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 IOCB #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  the wildcard characters * and ?  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 the user's responsibility.

```
Example: 1Ø DIM A$(1Ø25)
         2Ø BGET #5,ADR(A$),1Ø24
         3Ø A$(1Ø25) = CHR$(Ø)
```

This program segment will get 1Ø24 bytes from the file or device associated with file number 5 and store it in A$. Statement 3Ø sets the. length of A$ to 1Ø25.

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

Examples: 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 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>. 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 a 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 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 comma 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 FILE$
                 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

        Examples: 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.

5.1Ø 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: 1ØØ GET #Ø,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)...]|
                        | svar            |

Examples: 1ØØ INPUT X
          1ØØ INPUT N$
          1ØØ INPUT X,Y,Z(4)
          1ØØ INPUT ARRSTR$(5;)
          1ØØ PRINT "ENTER THE VALUE OF X"
          11Ø INPUT 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:

```
1ØØ PRINT "command>>  ";
11Ø INPUT #Ø, COMMAND$
```

The statement 'INPUT #Ø, COMMAND$' inputs a string from channel Ø (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: 1ØØ 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
                 1ØØ 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, as they have their own loading commands (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 specifier, no OPEN, or no CLOSE statement.

NOTE: The semi-colon and comma options are discussed in section 5.18, PRINT.

CAUTION: With most printers, LPRINT cannot successfully be used with a trailing comma or semi-colon. 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: 1ØØ 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,aeaxp2,filespec

Example: 1ØØ OPEN #2,8,Ø,"C:"
         1ØØ A$ = "D1:TEST.DAT"
         11Ø OPEN #2,8,Ø,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 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 exis-ting 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: 1ØØ 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...] |,|
                                 |,|          |,|

        Examples: PRINT
                  PRINT X,Y,Z,A$
                  1ØØ PRINT "THE VALUE OF X IS ";X
                  1ØØ PRINT "COMMAS","CAUSE","COLUMNS"
                  1ØØ PRINT #3,A$
                  1ØØ PRINT ,Ø:"$";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 semi-colon 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 comma or semi-colon 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 or 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) 1ØØ PRINT USING "## ###X#",12,315,7

        2) 1ØØ DIM A$(1Ø) : A$="## ###X#"
           2ØØ 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.

Examples:

Value	Format Field	Print Out
1	###	bb1
12	###	b12
123	###	123
1234	###	234
12	&&&	Ø12
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.

Examples:

Value	Format Field	Print Out
123.45	###.##	123.46
6	###.##	bb4.7Ø
4.7	##.##.	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 (Ø 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.

Examples:

Value	Format Field	Print Out
5216	##,###	b5,216
3	##,###	bbbbb3
4175	**,***	*4,175
3	&&,&&&	ØØØØØ3
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.

Examples:

Value	Format Field	Print Out
43.7	+###.#	+b43.7
-43.7	+***.*	-b43.7
23.58	-&&&.&&	bØ23.58
-23.58	-&&&.&&	-Ø23.58

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.

Examples:

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).

Examples:

Value	Format Field	Print Out
43.71	***.**+	*43.17+
43.71	&&&.&&-	Ø43.17b
-43.71	###.##+	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.

Examples:

Value	Format Field	Print Out
34.2	$##.##	$34.2Ø
34.2	+$##.##	+$34.2Ø
-34.2	+$###.##	-$b34.2Ø

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.

Examples:

Value	Format Field	Print Out
34.2	$$$$$.##	bb$34.2Ø
34.2	+$$$$$.##	+bb$34.2Ø
1572563.41	$$,$$$,$$$.##+	$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.

String Formats

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.

Examples:

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

the result: (498)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 semi-colon (;).

Example: PRINT #5; USING A$,B

will print B in the format specified by A$ to the file or device associated with file number 5.

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.

5.2Ø PROTECT

```
        Format:   PROTECT filespec

        Examples: PROTECT D:*.COM
                  1ØØ 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.)

```
        Format:   PUT #chan,aexp

        Examples: 1ØØ PUT #6,ASC("A")
                  2ØØ PUT #Ø,4*13
```

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...]

        Examples: 1ØØ READ A,B,C,D,E
                  11Ø DATA 12,13,14,15,16

                  1ØØ READ A$,B$,C$,D$,E$
                  11Ø 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) i. 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 in 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 "D2: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 used 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)  1ØØ DIM A$(1ØØ)
                :
              8ØØ RPUT #3,A$

          2)  1ØØ DIM X$(2ØØ)
                :
              8ØØ RGET #2,X$
```

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)  1ØØ DIM A$(1Ø)
              2ØØ A$ = "ABCDE"
                :
              8ØØ RPUT #4,A$

          2)  1ØØ DIM X$(1Ø)
              2ØØ X$ = "HI"
                :
              8ØØ RGET #6,X$
              9ØØ 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 25 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

        Examples: 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 in 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

Examples: 1ØØ 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.

5.3Ø XIO (X.)

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

Example: XIO 18,#6,Ø,Ø,"S:"

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

cmdno Command Number 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,Ø,Ø,"D:TEMP,CAROL"
33	DELETE	XIO 33,#1,Ø,Ø,"D:TEMP.BAS"
35	LOCK FILE	XIO 35,#1,Ø,Ø,"D:TEMP.BAS"
36	UNLOCK FILE	XIO 36,#1,Ø,Ø,"D:TEMP.BAS"
37	POINT	Same as BASIC POINT
38	NOTE	Same as BASIC NOTE
254	FORMAT	XIO 254,#1,Ø,Ø,"D2:"

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

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

filespec — string expression that specifies 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 1Ø2Ø through 1Ø3Ø test the file to see if it is segmented, so you can load in multi-segment files with this subroutine.

```
1ØØØ TRAP 1Ø9Ø
1Ø1Ø OPEN #1,4,Ø,"D:FILE.OBJ"
1Ø2Ø GET #1,L:GET #1,H
1Ø3Ø IF L=$FF AND H=$FF THEN GET #1,L:GET #1,H
1Ø4Ø START=H*256+L
1Ø5Ø GET #1,L:GET #1,H
1Ø6Ø FINISH=H*256+L
1Ø7Ø BGET #1,START,FINISH-START+1
1Ø8Ø GOTO 1Ø2Ø
1Ø9Ø IF ERR(Ø)=136 THEN CLOSE:RETURN
11ØØ PRINT "UNEXPECTED ERROR #":ERR(Ø);" AT LINE "; ERR(1)
111Ø 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

ABS	INT	RND
CLOG	LOG	SGN
EXP	RANDOM	SQR

Trigonometric Functions

ATN	RAD
COS	SIN
DEG	

String Functions

ASC	LEFT$	RIGHT$
CHR$	LEN	STR$
FIND	MID$	VAL

Game Controller Functions

HSTICK	PTRIG	VSTICK
PADDLE	STICK	
PEN	STRIG	

Player/Missile Functions

BUMP	PMADR

Special Purpose Functions

ADR	ERR	PEEK	TAB
DPEEK	FRE	POKE	USR
DPOKE	HEX$	SYS	

6.1 Arithmetic Functions

6.1.1 ABS

Format: ABS(aexp)

Example: 1ØØ AB =ABS(-19Ø)

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: 1ØØ C = CLOG(83)

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

6.1.3 EXP

Format: EXP(aexp)

Example: 1ØØ 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 2Ø.Ø855365.

6.1.4 INT

Format: INT(aexp)

Examples:
```
1ØØ I = INT(3.445) : REM I now = 3
1ØØ 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: 1ØØ L = LOG(67.89/2.57)

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

6.1.6 RANDOM

Format: RANDOM(aexp1[,aexp2])

Examples:
```
1Ø X = RANDOM(99)
1Ø Y = RANDOM(2Ø,3Ø)
```

The RANDOM function allows you access to a random number generator which does more than return a number between Ø and 1, as RND does. When used with one aexp (as in the first example), the value returned will be between Ø 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: 1Ø A = RND(Ø)

Returns a hardware-generated random number between Ø 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: 1ØØ X = SGN(-199) : REM -1 is returned

Returns a -1 if aexp evaluates to a negative number; a Ø if aexp evaluates to Ø, or a 1 is aexp evaluates to a positive number.

6.1.9 SQR

Format: SQR(aexp)

Example: 1ØØ PRINT SQR(1ØØ) : REM 1Ø is printed

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

6.1.1Ø 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.

```
1ØØ GRAPHICS 1 : REM set up screen
11Ø PRINT "Number to Manipulate> ";
12Ø INPUT #Ø, X : REM get the number
13Ø PRINT #6; ASC$(125) : REM clear screen
14Ø PRINT #6; "ABS.: ";ABS(X) : REM absolute value
15Ø PRINT #6
16Ø PRINT #6; "INT.: ";INT(X) : REM integer value
17Ø PRINT #6
18Ø PRINT #6; "SQRT: ";SQR(ABS(X)) : REM square root
19Ø PRINT #6
2ØØ PRINT #6; "CLOG: ";CLOG(ABS(X)) : REM common log
21Ø PRINT #6
22Ø PRINT #6; "NLOG: ";LOG(ABS(X)): REM natural log (ln)
23Ø PRINT #6
24Ø PRINT #6; "EXP.: ";EXP(X) : REM exponential (e^X)
25Ø GOTO 11Ø
```

6.2 Trigonometric Functions

6.2.1 ATN

Format: ATN(aexp)

Example: 1ØØ X = ATN(1.Ø)

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: 1ØØ 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

Examples: 1ØØ DEG
1ØØ RAD

These two statements allow the programmer in 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: 1ØØ 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)

Examples: 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 of 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 Ø is returned.

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

```
2 if N=Ø or N=1
9 if N>2 and N<1Ø
Ø if N>=1Ø
```

More Examples:

1)
```
1Ø DIM A$(1)
2Ø PRINT "INPUT A SINGLE LETTER:
3Ø PRINT "Change/Erase/List"
4Ø INPUT "CHOICE ?",A$
5Ø ON FIND("CEL",A$,Ø) GOTO 1ØØ,2ØØ,3ØØ
```

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

2)
```
1ØØ DIM A$(1Ø): A$="ABCDEFGHIJ"
11Ø PRINT FIND (A$,"E",3)
12Ø PRINT FIND (A$(3),"E",Ø)
```

Line 11Ø will print "5" while 12Ø will print "3". Remember, the position returned is relative to the start of the specified string.

3)
```
1ØØ INPUT "2Ø CHARACTERS, PLEASE:",A$
11Ø ST=Ø
12Ø F=FIND(A$,"A",ST):IF F=Ø THEN STOP
13Ø IF A$(F+1,F+1)<>"B" AND A$(F+1,F+1)<>"C"
        THEN ST=F+1:GOTO 12Ø
14Ø 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)

Examples: 1ØØ A$=LEFT$("ABCDE",3)

1ØØ 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: 1ØØ 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 Ø 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 routines illustrate the use of the LEN function:

```
1Ø A$="ATARI"
2Ø PRINT LEN(A$)
```

```
1Ø DIM AR$(3,Ø
2Ø AR$(2;)="ATARI"
3Ø 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'th character of svar, and is 'aexp2' characters long.

If aexp1 equals Ø 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 Ø), but if its value makes the substring go beyond the length of svar, then the substring returned ends at the end of svar.

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

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: 1ØØ 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
298 GOTO 100
```

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

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

6.4 Game Controller Functions

6.4.1 HSTICK

Format: HSTICK(aexp)

Example: 1ØØ IF HSTICK(Ø)>Ø 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 (Ø-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
 Ø 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. a.xp is the number of the paddle port (Ø-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(Ø)

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

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

6.4.4 PTRIG

Format: PTRIG(aexp)

Example: 1ØØ IF PTRIG(1)=Ø8 THEN PRINT "MISSILES FIRED!"

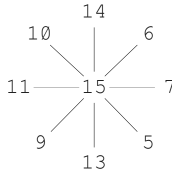
The PTRIG function returns a status of Ø if the trigger button of the designated paddle is pressed. Otherwise, it returns a value of 1. The aexp must be a number between Ø and 7 as it designates the paddle.

6.4.5 STICK

Format: STICK(aexp)

Example: 1ØØ PRINT STICK(3)

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



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: 1ØØ IF STRIG(1)=Ø 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(Ø)<Ø 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 (Ø-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
 Ø 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.

```
1ØØ GRAPHICS 5 : REM set up screen
11Ø COL=4Ø : RBM middle of screen
12Ø ROW=2Ø
13Ø COLOR 2 : REM drawing a cursor color
14Ø PLOT COL, ROW : REM plot cursor
15Ø FOR I=1 TO 15 : NEXT I : REM delay loop
16Ø IF STRIG(Ø)=1 THEN COLOR Ø:PLOT COL,ROW:REM don't draw point
17Ø COL=COL+HSTICK(Ø) : REM check for movement
18Ø ROW=ROW-VSTICK(Ø)
19Ø IF COL<Ø THEN COL=Ø : REM screen bounds checking
2ØØ IF COL>79 THEN COL=79
21Ø IF ROW<Ø THEN ROW=Ø
22Ø IF ROW>39 THEN ROW=39
23Ø FOR I=1 TO 25 : NEXT I : REM delay loop
24Ø GOTO 13Ø : 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(Ø,Ø)

BUMP accesses the collision registers of the Atari and returns a 1 (collision occurred) or Ø (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).

Valid BUMPs:
- PLAYER to PLAYER (Ø-3 to Ø-3)
- MISSILE to PLAYER (4-7 to Ø-3)
- PLAYER to PLAYFIELD (Ø-3 to 8-11)
- MISSILE to PLAYFIELD (4-7 to 8-11)

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

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,Ø
```

6.5.2 PMADR

```
Format:  PMADR(aexp)

Example: PØ=PMADR(Ø)
```

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)

Examples: ADR(A$)
          ADR(B$(5;))
```

Returns the decimal memory address of the string specified by the expression in parentheses. Knowing the address enables the programmer 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(13Ø)
```

The DPEEK functions is very similar to the PEEK function, except that it allows you to look 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(13Ø)+(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(Ø);" 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 any run-time error.

ERR(Ø) 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(Ø)=8 THEN GOTO ERR(1)
210 PRINT "UNEXPECTED ERROR #";ERR(Ø)
```

6.6.5 FRE

Format: FRE(aexp)

Examples:

```
PRINT FRE(Ø)
100 IF FRE(Ø)<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 (Ø) 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)

        Examples: 1ØØ PRINT HEX$(X+7)
                  2ØØ A$=HEX$(83)
                  21Ø PRINT "$";A$(3,4)
```

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

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

NOTE: no "$" is placed in front of the number.

6.6.7 PEEK

```
        Format:   PEEK(aexp)

        Examples: 1ØØ IF PEEK (4ØØØ) = 255 THEN PRINT "255"
                  1ØØ 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 e 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 e to 255. This function allows the user to examine either RAM or ROM locations. In the first example above, the PEEK is used to determine whether location 4~el!l (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

        Examples: POKE 82,1Ø

                  1ØØ POKE 82,2Ø
```

Although this is not a function, it is included in this section because it is closely associated with the PEEK function. This POKE command inserts data into the 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 Ø and 255. POKE cannot be used to alter ROM locations. In gaining familiarity with this command it 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 1Ø. 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: 1ØØ IF SYS(Ø)=Ø THEN SET Ø,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).

6.6.1Ø TAB

```
        Format:  TAB(aexp)

        Example: PRINT #3;"columns:";TAB(2Ø);2Ø;TAB(3Ø);3Ø
```

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 spaces 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: 1ØØ RESULT = 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 Ø 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 FRØ and FRØ+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.

BASIC XL statement:

```
-------------------
        PRINT HEX$(USR($68Ø,$3FFA,$2972))
```

USR routine at $68Ø:

```
--------------------
   FRØ = $D4
       *= $68Ø
       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 FRØ+1  ; store high byte
       PLA        ; get low byte of arg #1
       STA FRØ    ; store low byte
       PLA        ; get high byte of arg #2
       EOR FRØ+1  ; XOR it with high byte of arg #1
       STA FRØ+1  ; store result of XOR
       PLA        ; get low byte of arg #2
       EOR FRØ    ; XOR it with low byte of arg #1
       STA FRØ    ; 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 Ø into the low byte of the timer and waiting until it is greater than or equal to 6Ø.

Note: On PAL systems remember to use 5Ø; see line 2ØØ.

```
1ØØ GRAPHICS 2
11Ø PRINT #6; CHR$(125) : REM Clear Mode 2 area
12Ø PRINT : PRINT : PRINT
13Ø PRINT "COUNTDOWN TIME? ";
14Ø INPUT #Ø,X
15Ø POKE $14,Ø : REM set clock = Ø
16Ø PRINT #6; "TIME -";
17Ø   WHILE X>Ø : REM start the countdown
18Ø   POSITION 7,1 : REM get ready to print the new time
19Ø   PRINT #6; USING "##",X; : REM print time left
2ØØ     WHILE PEEK($14)<=6Ø : REM wait until a second has passed
21Ø     ENDWHILE
22Ø   POKE $14,Ø : REM reset the clock for the next second
23Ø   X=X-1 : REM decrement number of seconds left
24Ø   ENDWHILE : REM end of countdown loop
25Ø PRINT CHR$(253) : REM ring the bell
26Ø GOTO 11Ø : 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 graphics modes. The table below summarizes the 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 #Ø.

NOTE: aexp must be positive.

Graphics modes 8, 9, 1Ø, and 11 are full-screen display while modes 1 through 8 and 12 to 15 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 Ø in Direct mode, press <SYSTEM RESET> or type GR.Ø and press <RETURN>.

Gr. Mode	Mode Type	Cols	(split) Rows	(full) Rows	Num of Colors	
Ø	TEXT	4Ø	N/A	24	2	
1	TEXT	2Ø	2Ø	24	5	
2	TEXT	2Ø	1Ø	12	5	
3	GRAPHICS	4Ø	2Ø	24	4	
4	GRAPHICS	8Ø	4Ø	48	2	
5	GRAPHICS	8Ø	4Ø	48	4	
6	GRAPHICS	16Ø	8Ø	96	2	
7	GRAPHICS	16Ø	8Ø	96	4	
8	GRAPHICS	32Ø	16Ø	192	1 ½	
9	GRAPHICS	8Ø	N/A	192	16	
1Ø	GRAPHICS	8Ø	N/A	192	9	
11	GRAPHICS	8Ø	N/A	192	16	
12	TEXT	4Ø	2Ø	24	5	(XL/XE)
13	TEXT	4Ø	1Ø	12	5	(only)
14	GRAPHICS	16Ø	16Ø	192	2	(- " -)
15	GRAPHICS	16Ø	16Ø	192	4	(- " -)

7.1.1 GRAPHICS Mode Ø

This mode is the 1-color, 2-luminance (brightness) default mode for the ATARI Personal Computer. It contains a 24 line by 4Ø 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 the following method:

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

GRAPHICS Ø 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 Ø).

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 Ø, but are the same height.

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

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: 1ØØ GR. 1
                 11Ø 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	Ø	Orange
Lower case alphabetic	1	Light Green
Inverse upper case alphabetic	2	Dark Blue
Inverse lower case alphabetic	3	Red
Numbers	Ø	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 SETCOLOR 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.

7.1.4 GRAPHICS modes 4,6

These two 2-color graphics modes are split-screen displays and can display in only two colors while the other modes can display 4 and 5 colors. The advantage of a two-color mode is that it requires less RAM space. Therefore, it is used when only two colors are needed and RAM is getting crowded. These two modes also have a higher resolution which means smaller points than Graphics mode 3.

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 luminances, as mode Ø.

7.1.6 GRAPHICS modes 9, 1Ø, and 11

GRAPHICS modes 9, 1Ø, 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 mode. The main color is set by the background color, and the luminance values are determined by the information in the screen memory itself. Each pixel is four bits wide, allowing for 16 different values. 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 1Ø 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 1Ø 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
Ø	PCOLRØ	7Ø4	PMCOLOR Ø
1	PCOLR1	7Ø5	PMCOLOR 1
2	PCOLR2	7Ø6	PMCOLOR 2
3	PCOLR3	7Ø7	PMCOLOR 3
4	COLORØ	7Ø8	SETCOLOR Ø
5	COLOR1	7Ø9	SETCOLOR 1
6	COLOR2	71Ø	SETCOLOR 2
7	COLOR3	711	SETCOLOR 3
8	COLOR4	712	SETCOLOR 4

7.1.7 GRAPHIC modes 12 and 13

These are 5-color split-screen text modes.

The characters in mode 12 have the same height as in mode 0, but only four pixels get displayed instead of eight.

In mode 13 the characters are double the size of mode 0 characters, while only four pixels are displayed.

Since both modes display only four bits in each line of the character definition, the color of the activated pixel depends on the bit pair in the byte being addressed:

Bit Pair	Color	RAM Location
00	BAK	712
01	PF0	708
10	PF1	709
11	This depends on bit 7 of the byte.	

If bit 7 = 0, then use PF2 (at 710),
else use PF3 (at 711).

Each line in a character set definition (eight lines, one byte wide, form one character) can have different color combinations. Since bit pairs (one color clock) are displayed, the normal character set becomes unrecognizable. In order to use these modes, you should build a character set in which each character is half a letter and can be combined for display. Or build a 7x7 character set with a blank row and column between each character.

The characters displayed are only one half of the ATASCII set, depending on the value in location 756: 224 for uppercase, 226 for lowercase. The lower seven bits (0-6) are used for character data (range from 0 to 127), while the high bit is the color modifier (see table above).

7.1.8 GRAPHIC modes 14 and 15

GRAPHICS 14 is a two-color mode with half the horizontal resolution of GRAPHICS 8. Each screen line is one scan line high.

GRAPHICS 15 is a four-color mode with screen lines being one scan line high. Only the first two bits of a screen byte identify the byte color.

NOTE: Graphic modes 12-15 are not available from BASIC XL on 400/800 machines.

7.2 COLOR (C.)

```
        Format:   COLOR aexp

        Examples: 11Ø COLOR ASC("A")
                  11Ø 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 Ø 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 Ø through 2, the number can be from Ø through 255 (8 bits) and determines the character to be displayed and its color. (The two most significant bits determine 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 Ø or 1 (1-bit) and four-color modes require Ø, 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 Ø, 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 Ø. and when a GRAPHICS command (without +32) is executed, all of the pixels are set to Ø. 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: 1ØØ DRAWTO 1Ø,8

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: 15Ø 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 Ø to 255 for Graphics modes Ø through 2, a Ø or 1 for the 2-color graphics modes, and a Ø, 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:

```
1Ø GRAPHICS 3+16
2Ø COLOR 1
3Ø SETCOLOR 2,1Ø,8
4Ø PLOT 1Ø,15
5Ø DRAWTO 15,15
6Ø LOCATE 12,15,X
7Ø 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: 1ØØ 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 hue 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 (Ø,Ø), and extends to one less than the total number of rows (X-coordinate) or columns (Y-coordinate).

NOTE: PLOT aexp1,aexp2 is equivalent to:

```
      POSITION aexp1,aexp2 : PUT #6, COLOR
```

7.6 POSITION (POS.)

```
        Format:  POSITION aexp1,aexp2

        Example: 1ØØ 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)

```
        Formats:  PUT #6,aexp
                  GET #6,avar

        Examples: 1ØØ PUT #6,ASC("A")
                  2ØØ 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 the 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 Ø-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: 1ØØ SETCOLOR Ø,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 (Ø-4 depending on graphics mode)
aexp2 = Color hue number (Ø-15 --see the table below)
aexp3 = Color luminance (must be an even number between Ø and 14; the higher the number, the brighter the display. 14 is almost pure white.)

SETCOLOR aexp2	Color	SETCOLOR aexp2	Color
Ø	Gray	8	Blue
1	Gold	9	Light Blue
2	Orange	1Ø	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 Ø through 4. The Operating System (OS) has five RAM locations (COLORØ 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 Ø 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
Ø	2	8	Orange
1	12	1Ø	Green
2	9	4	Dark Blue
3	4	6	Pink or Red
4	Ø	Ø	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
Ø and all text windows	Ø	COLOR data actually determines	--
	1		--
	2		Character luminance
	3		Background
	4		Border
1,2, 12,13	Ø	the character to PLOT	Character
	1		Character
	2		Character
	3		Character
	4		Background, Border
3,5,7, 15	Ø	1	Graphics Point
	1	2	Graphics Point
	2	3	Graphics Point
	3	--	--
	4	Ø	Gr.Pt.,Border,Background
4,6, 14	Ø	1	Graphics Point
	4	Ø	Gr.Pt.,Border,Background
8	1	1	Graphics Point,Luminance
	2	Ø	Graphics Point,Background
	4	--	Border

7.9 XIO (X.) Special Fill Application

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

Example: 1ØØ XIO 18,#6,Ø,Ø,"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 (Ø) 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 (Ø) 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.

```
1Ø GRAPHICS 5+16
2Ø COLOR 3
3Ø PLOT 7Ø,45
4Ø DRAWTO 5Ø,1Ø
5Ø DRAWTO 3Ø,1Ø
6Ø POSITION 1Ø,45
7Ø POKE 765,3
8Ø XIO 18,#6,Ø,Ø,"S:"
9Ø GOTO 9Ø
```

7.1Ø SOUND (SO.)

Format: SOUND aexp1,aexp2,aexp3,aexp4

Example: 1ØØ SOUND 2,2Ø3,1Ø,12

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 voices available on the Atari (number Ø - 3).

aexp2 is the frequency (pitch) of the sound, and ranges between Ø and 255. The lower aexp2 is, the higher the frequency.

Aexp3 is a measure of the sound's distortion (fuzziness). Valid numbers are Ø - 14, even numbers only. A value of 1Ø 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 1Ø:

	aexp2	Note(s)		aexp2	Note(s)
HIGH	29	C		91	F
NOTES	31	B		96	E
	33	A# or Bb		1Ø2	D# or Eb
	35	A		1Ø8	D
	37	G# or Ab		114	C# or Db
	4Ø	G	MIDDLE C	121	C
	42	F# or Gb		128	B
	45	F		136	A# or Bb
	47	E		144	A
	5Ø	D# or Eb		153	G# or Ab
	53	D		162	G
	57	C# or Db		173	F# or Gb
	6Ø	C		182	F
	64	B	LOW	193	E
	68	A# or Db	NOTES	2Ø4	D# or Eb
	72	A		217	D
	76	G# or Ab		23Ø	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 29,31,35,40,45,47,53,60,64,72,81,91,96,108,121
90 DATA 128,144,162,182,193,217,243,256
```

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.

Chapter 8: PLAYER / MISSILE GRAPHICS

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") represent a portion of the Atari hardware totally ignored by Atari BASIC and Atari OS. Even the screen handler (the "S:" device) knows nothing about PMG.

BASIC 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" than are included in BASIC XL, read the Atari document entitled "Atari 4ØØ/8ØØ Hardware Manual" (Atari part number CØ16555, 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 stripe on the screen. Sounds dull? Consider: each player (there are four) may be "painted" in any of the 128 colors available on the Atari (see SETCOLOR for specific colors). Within the vertical stripe, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to Ø paints no color at all! That is, any Ø 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 Ø bits, in which case the background "shows through"). Using PMMOVE, you may then move this player to any horizontal or vertical location on the screen. To complicate:

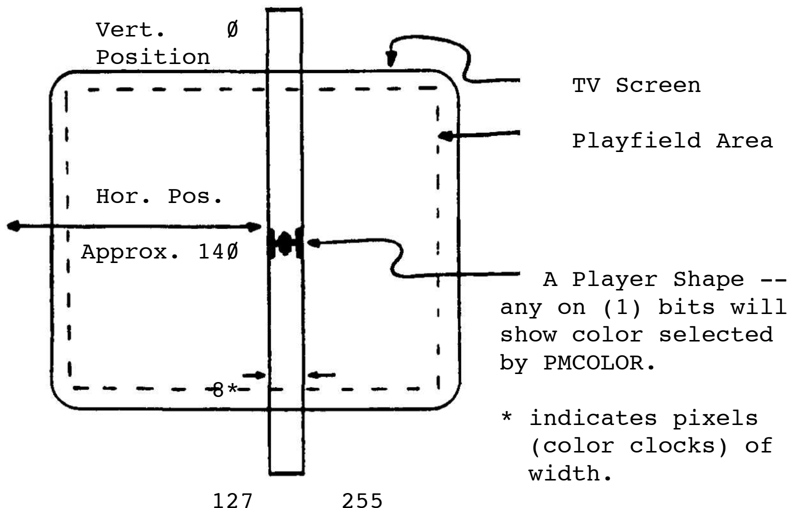
For each of the four players there is a corresponding "missile" available. Missiles are exactly like players except that:

1) they are only 2 bits wide, and all four 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 Ø through 3. Each player has a corresponding missile whose number is 4 greater then that of its parent player, thus missiles are numbered 4 through 7. In the BUMP function, the "playfields" are numbered from 8 through 11, corresponding to actual playfields Ø 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 Ø,Ø 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 Ø 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(Ø),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 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,Ø : 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 (luminace 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
```

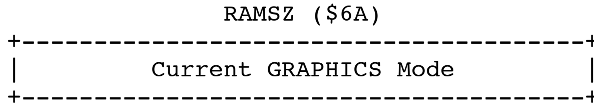
This statement is used to enable or disable the Player/ Missile Graphics system. The aexp should evaluate to Ø, 1, or 2:

```
        PMG.Ø   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.



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

```
                Double Line         Single Line
+1Ø24  +----------------------------------------+  +2Ø48
       |     Player 3       |                   |
 +856  |--------------------|    Player 3       |
       |     Player 2       |                   |
 +7ØØ  |--------------------|-------------------|  +1792
       |     Player 1       |                   |
 +641  |--------------------|    Player 2       |
       |     Player Ø       |                   |
 +512  |--------------------|-------------------|  +1536
       | MØ | M1 | M2 | M3  |                   |
 +384  |--------------------|    Player 1       |
       |                    |                   |
       |--------------------|-------------------|  +128Ø
       |                    |                   |
       |                    |    Player Ø       |
       |                    |                   |
PMBASE |--------------------|-------------------|  +1Ø24
                            |    |    |    |    |
                            | MØ | M1 | M2 | M3 |
NOTE: MEMTOP ($2E5) points  |    |    |    |    |
      to the bottom of the  |-------------------|
      missiles (PMBASE+384  |                   |
      in double line,       |                   |
      PMBASE+768 in single  |                   |
      line.                 |                   |
                            |                   |
                            |                   |
                            |                   |
                            +-------------------+  PMBASE
```

8.7 PMMOVE

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

        Examples: PMMOVE Ø,12Ø;1
                  PMMOVE 1,8Ø
                  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 Ø 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 semi-colon 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 screen to top of screen or vice versa) or should disappear as it scrolls too far up or down. SET 7,1 specifies wraparound, and SET 7,Ø 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 hit 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 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-or'ed with the current missile memory. This can allow the user to erase existing missiles while creating others.

```
Examples: MISSILE 5,4Ø,4
          MISSILE 5,4Ø,8
```

The first statement creates a 4 pixel missile at vertical position 2Ø. 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.

Examples: 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.

Example: 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
30 PMCLR 0 : PMCLR 4     : REM clear player 0 and missile 0
135 PMCOLOR 0,13,8       : REM a nice green player
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 128      : 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              : REM no more missile
440 GOTO 200             : REM and do all this again
450 REM
500 REM ********* THE DATA FOR PLAYER SHAPE *********
510 REM
520 DATA 153             : REM $99   *  **  *
530 DATA 189             : REM $BD   * **** *
540 DATA 255             : REM $FF   ********
550 DATA 189             : REM $80   * **** *
560 DATA 153             : REM $99   *  **  *
```

CAUTION: Do NOT put the REMarks on lines 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.

```
110 GRAPHICS 8            : 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
440 REM we better NEVER get to here !
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                   |*  *        *  *|
610 DATA $4812                   | *  *      *  * |
620 DATA $47E2                   | *   ******   * |
630 DATA $2004                   |  *          *  |
640 DATA $1008                   |   *        *   |
650 DATA $0C30                   |    **    **    |
660 DATA $03C0                   |      ****      |
```

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

The factor slowing this program the 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!

IMPORTANT NOTES

BASIC XL Cartridge Versions

The extended BASIC XL statements described in section 9.3 of this manual and the program demonstrating the use thereof described in section 9.4 will not work on BASIC XL cartridges with version numbers other than 1.Ø2 or higher. We are sorry about this, but the extensions "hook into" so many places within the cartridge that it is simply not practical to provide multiple versions of the code.

When you turn on your computer and enter the BASIC XL cartridge, there is a copyright notice which also specifies the version number of your cartridge. Check that version number. If it is not version 1.Ø2 or later, you have two options:

(1) Try to purchase a version 1.Ø2 or 1.Ø3 cartridge.

(2) Update the ROM in your cartridge to the latest version 1.Ø3.

Please note that current BASIC XL cartridges with version numbers 1.Ø2 or later are gold-plated (for longer and healthier life) and are beveled (for a better fit).

START PROGRAMMING!

9.1 THE BASIC XL RUNTIME PACKAGE

On the labeled side of your BASIC XL ToolKit diskette is a file called 'BASICXL.COM'. This file contains the BASIC XL RunTime Program. That program allows you to run BASIC XL programs without the BASIC XL cartridge.

9.1.1 How Does the RUNTIME Package Work?

The BASIC XL RunTime Program contains those portions of the BASIC XL cartridge which are used when programs are running. The program does not, however, contain any portions of the cartridge which are used to write new programs or edit existing programs. Thus, a program running under the BASIC XL RunTime Package can't perform such statements as LIST, ENTER, DEL, etc. Obviously, then, the BASIC XL cartridge is still required to develop programs.

The RunTime Program itself is just an Atari standard binary file which may be run under any Atari-compatible DOS, such as DOS XL or Atari DOS. The program may be run in any of three ways -- as an AUTORUN.SYS file, as a .COM file under DOS XL, or as an ordinary binary file using the 'L' option of Atari DOS. When the RunTime Program begins, it searches the disk in drive 1 (D1:) for the file AUTORUN.BXL. If that file is found, it is loaded into memory and run as if the command RUN 'D:AUTORUN.BXL' had been issued in response to the READY prompt. If the file AUTORUN.BXL is not present on the disk, RunTime will continually try to find it. You should eject your diskette, shut off power, and try again.

9.1.2 How Do You Use the RUNTIME Package?

The easiest way to use the BASIC XL RunTime Package is to perform the following steps:

1. Initialize a new disk and write DOS.SYS to it. You may use virtually any Atari-compatible DOS for this purpose. Note that DOSXL.XL (after being renamed to DOSXL.SYS) is compatible with RunTime.

2. Copy the file BASICXL.COM from the BASIC XL Toolkit disk to a file called AUTORUN.SYS on the newly initialized disk.

3. Copy the BASIC XL program you want to run to the new disk and name it AUTORUN.BXL.

4. Boot the disk thus created. If you have performed the previous steps correctly, your BASIC XL program will run automatically.

Whenever the disk you created above is booted, your program will run. If you have several programs you want to run with the RunTime Package and you don't want to dedicate several disks just to that purpose, you can simply put (or SAVE) some type of menu program onto the disk as AUTORUN.BXL and use it to select from other programs when the disk is booted. You are welcome to use the program MENU.BXL, described in section 9.2.1, for this purpose.

9.1.3 Statements that can NOT be used with RUNTIME

As we noted above, the BASIC XL RunTime Program does not contain those portions of the code from the BASIC XL cartridge which relate to program development. Any BASIC XL program which you want to use with the RunTime Package cannot use program development statements. If the BASIC XL RunTime Program encounters such a statement in your program, execution will stop with the message "Unimplemented statement in line XX", and you will be asked to hit the START key for a RunTime Restart (see below). The following is a list all BASIC XL statements illegal when using RunTime BASIC XL:

LIST	ENTER
NEW	DEL
RENUM	TRACE
TRACEOFF	LVAR

In addition, the following BASIC XL statements have slightly different meanings when using the RunTime Package:

DOS -- After this statement returns control to whatever DOS was booted, you can not return to BASIC XL or your BASIC program.

END -- This statement stops the running program and prompts the user to hit the START Key to do a RunTime Restart.

STOP -- This statement works exactly like END, but also prints the line number at which execution was ended.

9.1.4 Error Handling In RUNTIME BASIC XL

Errors which are TRAPped by the running program are treated exactly the same way as when using the BASIC XL cartridge. Errors which are not TRAPped are treated slightly differently, however. If an error is allowed to happen when no TRAP is active, an error message is displayed showing the line number where the error occurred, and the user is prompted to hit the START Key to do a RunTime Restart. The user is not allowed to view or change the program after an error as he could with the BASIC XL cartridge.

9.1.5 RunTime Restart

At various points above, we noted that under certain circumstances you may receive a message telling you to hit START to do a "RunTime Restart" (the message may indicate that RunTime will "Re-Run" a program). When this occurs, hitting START will cause RunTime to once again RUN the program file, AUTORUN.BXL. If your particular AUTORUN.BXL has chained to another program, the subsequent program is erased and all work not already written to file(s) is lost. (Note that RUN always closes all files, so at least no files are left dangling open.)

9.1.6 Incompatibilities

The only difference between RunTime BASIC XL and the BASIC XL cartridge which affects program execution is memory usage. Since RunTime BASIC XL is not in a SuperCartridge, it can't "save" memory like cartridge BASIC XL. For this reason, the BASIC XL RunTime Program takes up about 11 thousand bytes of code rather than 8 thousand bytes. If your BASIC XL program is extremely large, it may not run under RunTime BASIC XL.

9.2 BASIC XL Example Programs

Side one of your ToolKit disk contains ten programs written in standard BASIC XL which will, we hope, give you a feeling for the capabilities (and limitations) of the language.

Although the selection of programs is very broad, we certainly can not guarantee that you will find a program which answers all your questions about BASIC XL. In fact, perhaps we should begin by discussing some of the things which the example programs do not delve into.

First, we do not worry about the BREAK and RESET Keys. These programs are meant as examples for you, as a programmer or future programmer, to RUN and try. As such, we think YOU should be allowed and encouraged to stop a program at any time, see where it is at and what it is doing, and (our fervent hope) change it so it works better!

Second, we don't try to TRAP all disk errors, etc. The programs here all work properly if given properly formatted disks with the right data/programs (if called for). Again, our philosophy was to allowed you to explore the consequences of disk errors and guard against them in your own way. (And, truthfully, extensive I/O trapping in some of these programs is simply not necessary.)

Third, we do not get into any heavy math. For those of you who are into analytical geometry and its ilk, we apologize. Unfortunately, you are in a distinct minority when compared to those who want to use their machine for simple graphics and/or business applications.

Fourth, the descriptions of the programs (which follow immediately after this introduction) vary considerably in the depth with which they explore the workings of the code. Again, this is on purpose.

The most complicated of the programs (e.g., PICOADVENTURE and BLACK BOOK) are so large that even documenting each group of ten lines thoroughly would require a book several times the size of this manual. In these cases, we have tried to explain the principles behind blocks of code. You are encouraged (there's that word again,) to explore each and every line for its implications.

On the other hand, some of the programs are dissected in painstaking detail (e.g., MENU and GTIATEST). In some cases, we have chosen to be thoroughly simple to give beginners a chance to see the full workings of a program. In other cases, the thoroughness is dictated by the complexity of the subject. (Perhaps we are using a poorly documented feature of either BASIC XL or Atari's OS or hardware.) Mainly, though, we describe a program intimately because we want to get YOU in the right "track", thinking of properly structured programs, good error trapping, etc.

So much for the things we do not do in this ToolKit. What do we do? (We thought you'd never ask.)

If you are interested in graphics in general and games in particular, we turn your attention to SNAILS TRAILS, GTIATEST, CIRCLES, and (especially) LEM.

Into adventure games? Try PICOADVENTURE as a start on writing your own! (You might want to try playing and solving the game before reading the description.)

Want to learn more about how to talk to your disk drive? Look at CONFIG and DISKIO.

Interested in application programs? Want to learn how to construct random-access and/or keyed-access files? Look at BLACK BOOK.

Finally, MENU and MAKEAUTO are general utility programs. You will undoubtedly use them, but you may not need to understand them. But read about them anyway. The description of MENU, especially, is very detailed and gives some good hints on programming style.

A Commentary on Case -- In the descriptions which follow, we sometimes change a keyword or variable name to all upper case letters, despite the fact that the program listings will (as is usual in BASIC XL) show such names in mixed upper/lower case. This is done on purpose for emphasis only. You need not use upper case unless you have chosen Atari BASIC compatibility (via SET 5,Ø).

9.2.1 MENU.BXL

In most ways, this is the simplest program we will present in this section. MENU.BXL is simply a program which presents a menu of available BASIC XL programs and allows you to choose one of them to RUN. If you are an experienced Atari BASIC user, you have probably seen versions of this program floating around in magazines, user-groups, etc., for years. We think, though, that our version has some advantages which are worth discussing.

1070-1080 These lines set the tone for not only this program but, where possible, for all programs in this ToolKit. We really didn't need to initialize COUNT to zero, since BASIC XL guarantees that all variables start at 0.0 when a program is first RUN. But isn't this better? We both point out that we are using a variable named COUNT and that we know what its starting value should be.

Further, we could have coded line 1000 as

```
1000 Alpha = 64
```

but would that have any meaning to you? As clearly shows that ALPHA has a numeric value ATASCII value of the letter A.

1100 We chose the dimensions of FILES very carefully. There are 26 elements in the array because we won't allow more than 26 filenames in our menu. (That way we can select any program with a single letter, A to Z.) And each element has 14 characters because that is the maximum possible for a filename of the form "D:filename.ext". If you wish to allow disk drive numbers in your version of this menu, you will need to increase the second dimension here to 15.

1130 This POKE is documented in many books, including Mapping the Atari, from COMPUTE! books. A non-zero value turns off the cursor. A zero value turns it back on.

1240 Did you remember that an OPEN in mode 6 is actually an OPEN of the directory? Good. For all intents and purposes, this OPEN will cause subsequent INPUTs to read the same data you see when you give a DIR command. Try it. Type in

```
DIR "D:*.BXL"
```

and see what is displayed. (Yes, yes, the quotes aren't really needed. We Know, thanks.)

1250 Sometimes, in our zeal to avoid GOTO statements, we have gone to great lengths in these example programs. This is a good instance of such a great length. We read the first file name from the directory here solely because we want the WHILE loop that follows to look neat. Ah, don't knock it. It works.

1260 We begin the promised WHILE loop. Note how we ensure that we won't get more than 26 names. We check the second character for a space because the only line of the directory where it is not a space is the line noting the number of free sectors (which is not coincidentally, the last line of the directory).

1270-1310 We develop the name which will be held in the string array, File$. First, we count this as a valid name. Then we find out where the first blank is after the first letter of the filename is.

Example: for the file "MENU.BXL", the directory listing is

```
        * MENU     BXL 008
```

or similar, where the '*' means, the file is PROTECTed and the '008' is arbitrary. Here the FIND function would tell us that the value of BLANK will become 7, the blank after the 'U' of 'MENU'. Line 1290 is necessary in case the file has 8 letters in its name (the blank found will then be the one between the extension and the number of sectors).

In line 1300, we play a trick that works neat and sweet in BASIC XL (and also in Atari BASIC, but we had to brag a little): As long as you are moving characters "down" in memory (think of that as moving them left in a printed string), you may overlap your string assignment without error! This line, then, strips off the first two characters and all characters from the blank on. Bingo.

Finally, in line 1310, we actually put the name into the array. Note the form it takes: "D:filename.BXL",
"filename" may have from 1 to 8 characters.

1328-1348 This is Just a bit tricky. Since we want our menu to be able to hold 26 names, we can't simply list them straight down our 24 line screen. We must put them two to a line. The expression COUNT&1 (where '&' is BASIC XL's 'bitwise and' operator)

effectively checks whether COUNT is even or odd. If the COUNT is odd, we will put the name at horizontal (X) position 7. If it is even, we will put it at X-position 22.

The vertical position is also obtained through a little magic. To see why it works, try various values for COUNT and observe what Y value results. We will start you off:

If COUNT is	...	Y will be
1	...	3
2	...	3
3	...	4
26	...	15

Okay? Then line 134Ø is easy. We simply POSITION ourselves at the place we have calculated and print an indicator and the name. But just what is that indicator? Remember, ALPHA is one less than the ATASCII value of the letter 'A'. So if COUNT is 1, PRINTing CHR$(Alpha+Count) will produce the letter 'A' on the screen. Similarly, a COUNT of 2 will produce a 'B', etc. Now you know why we chose the value for ALPHA which we did.

135Ø-137Ø Here we simply get the next line from the directory and go back to the top of the WHILE loop. If it isn't a name (i.e., if it is the free sectors line) or if we already have 26 names, the loop will halt and fall through to the CLOSE of line 137Ø. We are then done with the directory.

141Ø This is the best way to get a single keystroke on an Atari computer. OPEN up the keyboard ("K:") and GET a Key (as in line 144Ø). Sure, you can do it with PEEKs and POKEs and whatever, but why bother? (Exception: if you don't want to wait for the key, you will have to use at least one PEEK.)

142Ø and 151Ø This is an "endless" WHILE loop. We could have achieved the same thing by eliminating line 142Ø and changing 151Ø to read GOTO 143Ø. But that's terribly ugly! As well as being poor structured programming style.

143Ø-147Ø We ask the user to press a key, get the key from the keyboard, and strip it of extraneous bits. Ummmm..." extraneous bits"?

By doing a bitwise and (&) of KEYPRESSED with $5F (that's 95 decimal or Ø1Ø11111 binary), we have removed the uppermost bit (bit 7 -- which would indicate inverse video) and also bit 5 (which

distinguishes upper case letters from lower case). So no matter what kind of letter the user pushes, we see an upper case, non-inverse video character.

Now, if it truly was a letter, subtracting ALPHA from it will convert it into the range of 1 to 26. Funny thing how the elements of our string array are numbered from 1 to 26. Do you think that's a coincidence? (If so, we've got some beachfront property in Nevada we'd like you to invest in.)

So, in line 146Ø, we validate that the letter chosen is in the range we have filenames for. (If it isn't, we skip to line 15ØØ, the ENDIF, and go through the WHILE loop again.) Then we show the user what filename he/she chose. Just to keep them happy while...

148Ø-149Ø Line 148Ø illustrates the proper use of a TRAP in a well structured BASIC XL program. You should always TRAP to the last line of a loop or condition. Here, if we get an error in line 149Ø, we want to go back and ask for another menu selection. Voila.(Exception: Sometimes you will want to have a central routine for handling TRAPped errors. That's a good idea, but beware of leaving WHILEs, GOSUBs, etc., sitting on the RunTime stack.)

And, at last, we get to use this program as it was intended. We actually RUN the program requested by the user. Note that since we PRINTed the name in line 147Ø it's hard to make a mistake here. But a diskette failure (bad sector, etc.) could trigger the TRAP when the file doesn't load properly. We emulate the Boy Scouts: Be Prepared.

9.2.2 SNAILS

If you read '3Ø Days to Understanding BASIC XL' (or, better yet, work your way through it), you will probably remember Chapter XXIX and a arcade game program called SNAILS' TRAILS. This game can give you a real feeling of historical perspective!

By today's standards, SNAILS' TRAILS is a simplistic game with marginal video appeal. A short five or six years ago, though, a very similar game called SURROUND was one of the hot sellers in the Atari 26ØØ VCS market. And, as recently as the time of the Disney movie "Tron", the "light cycles" played a variation on the same game.

Anyway, since this game has been overdone already, why are we rehashing it on this disk? Truthfully, because the version in our tutorial was written using only the statements presented in that book, and we wanted to show you what just a few added statements could do to a BASIC XL program. The result is a well structured and even readable program.

In the description which follows, we will not explore those parts of the program which are the same as the version shown in the book. (Note that the line numbers do not match those in the book. Sorry about that, but there are enough differences that they couldn't have been identical, anyway.)

18Ø In the book, we had two variables (SCOREØ and SCORE1) to keep track of the players' points. Here, we use a two element array. We'll show why below.

26Ø Isn't this easy to understand? You can translate this into English as follows: "As long as neither player has scored 1Ø points, keep playing!"

29Ø and 34Ø In the original, the COLORs are different. We changed them because it makes it easier to flash one of the slime trails (line 8ØØ).

49Ø-5ØØ The main movement loop translates to English pretty well, also: "While neither player has hit anything." Then, since we aren't driving this loop with FOR MOVE... anymore, we have to bump the MOVE number. The only place MOVE is used, though, is in line 69Ø, as the frequency value in a SOUND statement. But SOUND won't let us use a value greater than 255 for frequency, so after bumping MOVE we limit it to an 8-bit value.

You say you don't understand how bitwise-and (&) works after reading the brief description in the

reference manual (section 2.2.1)? We won't go into a lot of detail here, but let's show what happens in line 5ØØ as the value of MOVE increases. (In the binary notations below, we show only 12 bits instead of the 16 bits which BASIC XL always works with. The upper four bits are always zero in this example, though, so they can be ignored.)

```
    MOVE = 3 decimal, binary ØØØØ ØØØØ ØØ11
           bitwise and with ØØØØ 1111 1111
                            --------------
              binary result ØØØØ ØØØØ ØØ11
               (decimal value of 3)

  MOVE = 243 decimal, binary ØØØØ ØØØØ ØØ11
           bitwise and with ØØØØ 1111 1111
                            --------------
              binary result ØØØØ 1111 ØØ11
               (decimal value of 243)

    MOVE = 2 decimal, binary ØØØØ ØØØØ ØØ1Ø
           bitwise and with ØØØØ 1111 1111
                            --------------
              binary result ØØØØ ØØØØ ØØ1Ø
               (decimal value of 2)
```

Do you see what happens? When the value of MOVE becomes greater than 255, the bitwise-and effectively subtracts 256 from it. In fact, we could have coded line 5ØØ thus:

```
5ØØ Let Move=Move+3:If Move>255 Then Let Move=Move-256
```

But using the bitwise-and is faster yet, once you understand bitwise operators, just as easy to understand.

And, as long as this explanation is too long already, let us note that we could have achieved the same effect by using these two lines instead:

```
5ØØ Let Move=Move+3
69Ø Sound Ø,Move&255,1Ø,Volume
```

However, the SOUND statement is inside a tight loop, and placing the bitwise-and in the loop would slow it down a bit.

6ØØ-65Ø There's nothing really very different from the book version here except the order of the statements. We thought this scheme is more readable. We hope you agree.

76Ø Why didn't we just code this line as follows?

```
76Ø If BangØ <> Bang1
```

Because the values of BANGØ and BANG1 could be 1, 2, or 3, depending on who hit what. Using NOT BANGØ and NOT BANG1 converts all values to a boolean (zero or one) condition, which is more easily testable.

If you prefer positive logic, you could change 76Ø and all following references to BANGØ and BANG1 to this:

```
76Ø BangØ=Sgn(BangØ) : Bang1=Sgn(Bang1)
761 If BangØ<>Bang1
```

(Recall that SGN() of any positive number is one, as we want here.)

77Ø See line 76Ø, above. This line looks strange, so let's translate it into English: "Bump the score of the player who did not get banged by one." Still confused? Then substitute the following for line 77Ø:

```
77Ø If Bang1=Ø : Bang(1)=Bang(1)+1
771 Else : Bang(Ø)=Bang(Ø)+1 : Endif
```

But, if you're willing to struggle with the logic a bit, you will conclude that our original line 77Ø achieves exactly the same result with less code.

8ØØ Same thing again. Remember, NOT BANGØ is a logical expression, so it can only take on numeric values of zero and one. Cute?

89Ø Another case of a logical expression being used to derive a numeric value. If SCORE(Ø) really is less than SCORE(1), then WINNER will receive a value of one. Otherwise, WINNER will be set to zero.

Technical note: Most languages support the notion of TRUE and FALSE logical expressions. Unfortunately (?), many restrict their use to places where a conditional test is being made. However, BASIC XL, in common with many, many other (but not all!) dialects of BASIC, allow you to treat TRUE and FALSE as numeric values. Be careful, though, in some Microsoft (and other?) BASICs TRUE is given a value of minus one (-1) for reasons which are mired in history. (n.b.: BASIC is not the only language, which allows logical expressions to produce numeric values. C and some versions of Fortran allow similar usages.)

910 and 930 See how neatly we Can use WINNER now that we know it has a value of either zero or one?

980 In English you read this line to say: "As long as neither joystick trigger is pushed, keep looping."

9.2.3 PICOADV

In addition to being the longest program on the ToolKit disk, PICO-AOVENTURE is also the oldest. It was one of the first major programs we wrote for BASIC A+ (back in 1981-82) and is given here with only minimal modifications, even though it could probably use many of BASIC XL's new statements to advantage. Nevertheless, PICO-ADVENTURE (which name was intended to imply that it is smaller than a Micro-Adventure) is still a reasonably well-written, well-structured program which deserves more than a cursory glance.

For all of its size , PICO (as we shall call it from now on) only uses about half of the memory available when you use BASIC XL with DOS XL. If you feel so inclined, you may retain the structure of the program, replace room descriptions and object actions, and thus produce your own adventure. Nothing could please us more. In fact, we would love to see your results.

One last warning before we start looking at PICO a block at a time: Why don't you RUN and play it before reading this section. In studying the program, you will of necessity see the secrets of the game, which will destroy the pleasure you will get from winning (or losing) gracefully.

Because this program is so large, the best we can do is describe blocks of lines. We will delve into detail only when we feel that reading the program lines within the block won't give you enough understanding of their actions.

Finally, we present this program in execution order (not line number order), because you need an understanding of some of the subroutines before the main line code makes a lot of sense.

1ØØ-119 We use the question mark (?) abbreviation for PRINT a lot in this program. It makes the listing smaller and allows all lines to fit in the bounds of a 12Ø column printer. If you are going to list this program to an 8 inch (nominal 8Ø column printer, the ends of some lines will either wrap or get cut off (depending on how your printer works). If your printer has elite (12 characters per inch) or condensed (usually about 16 characters per inch) print available, we recommend that you set it in one of those modes before listing the program. All program lines will list on one printer line in condensed mode. Almost all will list properly in elite mode. (Note: an easy way to put your printer in one of these modes which works with most printers is to put its control or escape code

sequence right into a REMark line at the beginning of the program.)

We also use some imbedded screen control characters in our quoted strings, something we do not normally do with programs intended to be listed by YOU, our customer and reader. Again, we felt justified using them here (instead of using a CHR$() sequence), because they save so much room. We apologize in advance if they do funny things to your printed listing.

150 We put the initialization code up out of the wax as a subroutine so that the program looks better.

8000-8100 Primary initialization. Some variables used as constants, subroutine addresses, or counters are assigned here. Various strings and arrays are dimensioned. Some sizes are arbitrary and/or could be made bigger for a more complex adventure (one that understands more nouns or verbs). Ones that are carefully selected include VS$ and NS$, which are just long enough to hold a prefix character and a three-letter verb or noun. (See lines 1200 to 1300 and next paragraph.)

8110-8190 We build up the vocabulary lists for the verbs and nouns. Each entry in a list consists of a prefix character (CHR$(155), but any value from 128 to 255 would have worked), a three letter name, and a single byte which holds the verb or noun number associated with this name. Note that the name's number corresponds to the last two digits of the DATA statement from which the name was READ. For example, the first two entries in NOUNS$, the noun vocabulary list, would look like this (where a number in brackets indicates a byte with that value):

```
[155] L I C [1]
[155] M O S [1]
```

Also, as we build the noun vocabulary, we are setting up the WHERE() and SHOW arrays. A noun's entry in SHOW() tells the "visible items" routine whether to show it or not. The entry in WHERE() tells where the item (noun) is located, according to the following table:

```
If WHERE(noun-number) is ... noun is located
            less than 0 ... gone forever
                      0 ... with adventurer
                   1-99 ... in that room number
        greater than 99 ... still hidden
```

8800-8999 The DATA statements which define the verbs (88xx) and nouns (89xx). In theory, then, you could have up to 99 verbs and 99 nouns, each with one or more synonyms. Synonyms are simply listed one after the other on the same DATA line, the last one terminated by an asterisk. Tho first synonym is the one shown by the command line echo, inventory list, and visible items list, so it is spelled out completely. As noted above, nouns also have their initial WHERE and SHOW values listed here. The last entry in each table is terminated by a pound sign (#).

160 Getting a key one at a time from the "K:" device is still the best way. Much easier and more readable than PEEKs and POKEs.

920 This is kind of a cute trick. Rather than print out a special starting location message, etc., we simply tell our movement subroutine (starts at line 7000) that we are in room number 7 and that the user just asked us to go west. We also note that room number 3 is West of the current room. Then we GOSUB to do the movement and (PRESTO!) everything comes up right for somebody who just walked into Room 3! (Much of this will become clearer later...Keep reading.)

1050 Again, we could have coded the subroutine at line 6000 right in-line here (since it is called only once), but this makes the program so much more readable. Besides, wait until you see what that subroutine does.

6000-6199 Special actions processing. In many adventure games, including this one, certain actions must take place at certain times and/or after a particular number of turns have passed since some other event. For example, in PICO, the effect of eating the mushroom wears off after 4 turns. This time period is counted down in the variable CRAZY, and lines 6010 and 6030 reflect this. Three other such variables, CHARM, TORCHFIRE, and HUNGRY are similarly accounted for here. Note that, in lines 6100 to 6103, those counters are never allowed to become less than zero. One of them, HUNGRY, cycles from 29 down to zero, over and over.

1110-1190 This is our get-a-command routine. We only allow a few characters to get through. All others are ignored. Note that the variable OK is used both as a flag and as a counter to the current character within RESPONSE$. If the user hits RETURN (line 1130) we get out of the WHILE loop by simply setting the OK flag to zero. Cute.

In line 1140, we only allow back spacing to the beginning of the command typed in so far. And we special case inverse video space (KEY=160) for safety's sake. Finally, when we have masked all characters to be upper case and non-inverse video, we make sure that the user typed an alphabetic character. And, last but not least, we limit the user's response to 15 letters. That's more than enough (as we will see).

1200-1290 We parse the user's response into verb and noun parts. Or at least we try to. Lines 1215 and 1250 strip off leading spaces (line 1210 guaranteed that RESPONSE$ would contain at least something or these lines might generate errors). The verb is presumed to start at the first non-blank character and continue to the next following blank. (If there isn't a verb, we go back to line 1000 and get another response.) The noun is assumed to be everything after the blank(s) which follow the verb.

Again, note how the search variables, VS$ and NS$, were carefully dimensioned to 4 so that they could hold our separator character and the three significant letters of a verb or noun. (Do you see how you could easily increase the number of significant letters in a PICO vocabulary word?)

Lines 1280 through 1290 allow for the special case of a single letter response indicating a direction to take. Can you see how easy it would be to add Up and Down to our list of valid directions?

In any case, we come out of this block with the variables NOUN and VERB holding numeric values which represent the action requested by the user. (See the explanation of lines 8000-9000 for details on what the numbers mean.)

1300-1330 Pretty simple. If we didn't find a valid verb, say so. Ditto for a noun. Do you see why we tacked "   is." onto RESPONSE$ in line 1210? If the user tells us to EAT GORP, the variable NOUN$ will be set to "GORP is." Maybe a little too tricky?

1400-1514 One of the neatest things about PICO is that it tells you what It thinks you said. We've played adventures where we typed in "GET SNARE" only to have it tell you "You got it, but it bit you. You're dead." How were we supposed to know that SNA meant "snake" to that game? In PICO, if you type in "NIB MOS", the game will tell you that it is trying to "EAT LICHEN". A nice touch, we think.

152Ø and 2ØØØ-212Ø There is a bug in BASIC XL which has existed since the earliest versions of Atari BASIC. We're afraid to fix it, because there may be programs which depend on its action! Anyway, the bug is simple: if you GOSUB to a non-existent line, the GOSUB is pushed onto the run-time stack before the error is discovered. Subsequent RETURNs can then end up going back to the wrong place(s). We avoid the problem here by GOSUBbing to a known good line (2ØØØ).

Then, at line 21ØØ, we play a little bit of magic. Do you see what line number we try to go to? If the user requested verb number 7 and noun number 2, we will try to GOTO·line 17Ø2Ø. Suppose, though, that line 17Ø2Ø doesn't exist (as it doesn't in PICO). Then the TRAP 211Ø is activated and we GOTO line 17ØØØ instead.

Why? Well, as PICO is written, trying to BURN MUSHROOM will give us verb 7 and noun 2. Since line 17Ø2Ø doesn't exist, we end up at line 17ØØØ, where OK is set to NO so that the message, "That didn't make sense!" will be displayed. Since most items won't BURN, this provides a convenient method of processing all such non-productive requests the same way.

16ØØ-161Ø This ELSE clause was started by the IF of line 151Ø. The direction abbreviations (N,E,S,W) produce verb numbers of less than zero (-1 through -4). Once you understand the routine at line 7ØØØ, this part becomes easy.

7ØØØ-7Ø5Ø The variables NORTH, EAST, SOUTH, and WEST are already set up by the time we get here (we'll see how in a moment), so all these lines do is put the proper value into GO. And what's a "proper" value? Keep reading...

71ØØ-719Ø When we get here, GO can have one of four meanings:

If GO is	... we will
negative	... drown
zero	... do nothing (direction unavailable)
1-99	... go to that room number
1ØØ+	... do a special action

The "special action" trick is a neat one, uniquely available in BASIC XL and its brethren, because GO actually designates the line number of the subroutine to GOSUB to perform the action!

7200-7390 And here is where we get the values that end up in GO! After we have moved to another room (HERE=GO in line 7160), or even if we haven't, we RESTORE to the proper room description (line 7200, also uniquely BASIC XL, etc.). We READ in the lines of description (an equal sign on the end of a line indicates more to follow) and then, in line 7300, READ the four directions, NORTH, EAST, SOUTH, and WEST.

Isn't this neat? Look at lines 30160 to 30165. Just by the line numbers, we know that this is the DATA for room number 16 (30000+16*10). The description is 3 lines (each in quotes) long. And the connecting rooms are 15 to the NORTH, 12 to the WEST. But look at the "connections" for SOUTH and EAST, both get a value of 30164. That means that, if the user asks to go SOUTH or EAST from this location, line 7130 will end up doing a GOSUB 30164. So line 30164 is actual executable code (not more DATA) and the poor guy gets zapped by a truck.

Examine some of the other DATA statements in this range. Note how easily we drown adventurers (connecting "room number" of -1) or bar them from proceeding (connection values of zero). It's downright easy to add rooms and conditions to this game!

1800 Believe it or not, this is the "end" of the program. Everything after here is a subroutine. Ain't structured programming neat? Yeah? Then why didn't we use an endless WHILE loop instead of this old-fashioned GOTO? Sigh.

With all the main-line code described, we proceed to some of the subroutines not yet discussed.

7500, 7600, 7700 Three useful little routines, for when the user asks for something not available (7500), uses something he doesn't have (7600), or dies gracefully (7700).

7800 Four entry points provide delays of 1, 2, 3, or 4 seconds, thanks to the clock ticker in location 26.

7900 We display the stuff lying around on the ground. Remember, even if something is located in this room, we don't tell the user unless its SHOW() flag is true. This little nastiness makes PICO harder than it would otherwise would be. You could expand this in your own game(s) as you wished.

Finally, we get to the VERB and VERB/NOUN action routines. Remember, a VERB/NOUN action starts a line 10000+1000*VERB+10*NOUN. With this formula (and with

line numbers 10000 to 29999 available) you can have 20 different verbs (if they are numbered starting at zero) and 99 nouns. Changing the multipliers (e.g., make-it 500*VERB+20*NOUN) could change those ratios and/or make more lines available for particular actions.

Also recall that a VERB (alone) action starts at 10000+1000*VERB, and VERB/NOUN actions specified end up at those VERB alone lines.

We do not want to (nor do we feel we need to) devote the space to a complete description of all the possible actions. Instead, we will single a few out and leave the rest to you as an exercise.

13000-13173 These are the actions taken when the user asks to LOOK at something. Let's see what happens when he/she asks to LOOK JUNKPILE.

First of all, if Golem isn't in the right room (line 13170), how can we look at it? The rest of the responses depend on the value of JUNKCNT, which was initialized to 3.

If JUNKCNT is not zero, then we let the user find something in the pile. What he/she finds depends on the value of JUNKCNT (line 13172). The item(s) thus found (item numbers 9, 3, or 8, in that order) are made visible by giving them a location in the WHERE() array (line 13173). Recall that all three of these items received an initial location of 100 (hidden) in the DATA statements of lines 8900 to 8999. Note that changing WHERE() is all that is needed to cause the visible items print routine (lines 7900-7970) to make it show up.

If JUNKCNT is zero (all three items have been found), then we are sent off to line 13000, just as if we had typed LOOK BOAT (which would cause the routine at line 13150 to be executed, if it existed).

Line 13000 starts with a cute trick: If the user typed in just LOOK, the program pretends he/she really wanted LOOK PLACE. 13001 is pretty straightforward if you know how to read it: "If the Golem isn't carrying the requested object WHERE(NOUN) isn't zero) and if the object is not in this room (WHERE(NOUN) is not the same as HERE), then we can look at it, so ask the dummy HOW we can do it."

Finally, line 13002 simply gives a nice bland message about the object. If the user typed just LOOK (with no noun), then the message refers to "this place." Not exciting, but it works.

16000-16169 Almost every adventure you try will have some sort of secret word or phrase which you must SAY to unlock the mysteries. In PICO, we hint at that ability by providing you with a MAGIC LAMP (in the Junkpile) and putting a message on the billboard which has a message in quotes, usually a dead giveaway that the phrase ("A LAD IN BAGHDAD" in this case) is the sought after magic word(s).

In fact, if you use the command SAY A LAD IN... before you get the lamp, we even give you a clue (line 16160) that you need something else before the magic works.

But all of this is in vain. We borrowed a page from Sesame street and put the "fix" in: all you get for all your trouble in this game is a peanut butter sandwich. (To add insult to injury, it doesn't even fill you up! Of course , that's because the "I'm hungry" message is trying to make you eat the mushroom, another trick cadged from a children's story.)

That's about it for PICO. (Isn't it enough?) We hope you will turn it into your game and share it with us all.

9.2.4 LEM

This program is yet another incarnation of the classic lunar lander game. The principles of this game haven't changed, since people first started using computers to have fun, even if they were using time-sharing on mainframes and mini-computers back in those prehistoric days. For example, we have a book (fashioned from clay tablets, we think) dated 1975 (A.D.!!!) and called 'What to Do After You Hit RETURN or P.C.C.'s First Book of Computer Games' which includes no less than two different lunar lander programs. They were played on H.P. minicomputers with teletypes (you know...at a maximum of 1Ø characters per second, and no graphics).

So what's different about this program, and why should we discuss it? Well, it's written entirely in BASIC (big deal, so were those 1975 gems). And it uses pretty graphics (that's a little better). And it runs in real time (whazzat? Impossible!).

To play this game, plug a joystick into socket number 1 (STICK(Ø) in BASIC) and RUN the program from disk. You can play on two levels, beginner or advanced, but we recommend you to try it first as beginner, so simply push the joystick button. You will be presented with a moonscape, a bar at the left showing your remaining fuel, a landing pad (which will blink), and an odd-shaped ship (complete with antennae, legs, etc.) which you will (try to) control.

To move the ship left or right, simply push the joystick left or right. Be careful! The effects of such pushes are cumulative with time. Gentle taps in the appropriate direction work best.

To fire your retro-rockets, push the joystick button. If you do nothing further, you will probably crash (albeit perhaps slowly). That's because there are six possible settings on the LEM. You increase thrust by pushing forward on the joystick, decrease by pulling back. Need we tell you that greater thrust eats fuel faster? (If you run out of fuel, you run out of thrust. Need we tell you the results?)

If you manage to land (or- even crash) on the landing pad, you get points. Too fast a landing results in a crash. A landing of moderate speed gives you a bouncing good time. And a near perfect gets you applause and cheers from the crowd. (Which ignores the fact that sound doesn't carry in the vacuum on the Moon. Oh, well, maybe they're back on Earth?) You get 25Ø points for a great landing, 1ØØ points for a bounce, and credit for remaining fuel. You also get bonus points for the actual speed of your landing and the narrowness of the pad you landed on.

It's a good game. We've played it many, many times, and it's still a real challenge to score over 25ØØ points in five landings (a standard game) on the Advanced level. Before perusing the explanation of the workings which follows, why not try it yourself a few times.

This is a big program, but it is very well self-documented (with both REMarks and self-explanatory variable names). As with PICOADV (section 9.2.3) we will discuss this game in blocks, concentrating on the non-obvious features.

1ØØØ-129Ø After waiting for the player to let up on the joystick button, we present him/her with a menu and some brief instructions. LEVEL is set to zero for a beginner and one for an advanced player. Notice how we position the arrow, basing it or, the value of LEVEL. Also note how, after detecting the fact that the joystick has been pushed, we wait for the stick to come back to the center before continuing the loop. If we didn't do this, the arrows would flick back and forth from one level to the other almost too fast to see. (Try it yourself. Remove line 118Ø, and see what happens.)

13ØØ-176Ø Mostly simply initializing various arrays and strings. We will show later how these variables are used. Note how we choose one or the other set of DATA in lines 17ØØ to 172Ø, depending on the level of the player. You could have more than two levels here, if you wished, by adding choices to the initial menu and DATA for the acceleration values.

Speaking of which: The first acceleration number is the force of gravity. In other words, the positive attraction inviting you to crash into the rocky surface. The other six numbers are the acceleration values produced by the various thrust settings. Note that, on advanced level, the lowest thrust doesn't even cancel the pull of gravity. You can play with these numbers, but the game works pretty well with the values shown.

18ØØ-183Ø These are some critical constants used throughout the game. We need to discuss them just a little.

A POKE of any value to HITCLR clears the collision registers (see "Mapping the Atari"). The YSIZE is the height of the active playing area (in pixels) in GRAPHICS 7+16. If you wanted to play with GRAPHICS 15+16 (available only on XL machines), you could change this.

The lander spaceship (LEM) uses player Ø. Its flame (from the thrust) uses player 1. They are offset a

bit (from the base addresses of their respective players) to account for differences in their sizes. If you changed the appearance of the ship, you could adjust just ADRLANDER and ADRFLAME, and all would still work.

LANDER and FLAME are established just to save time in the tight loops later on.

We display the fuel remaining using player 2. The "+ 32" and "+159" values are empirical--they match the line to the size of the playfield nicely.

1890,3750 The limits of the once-per-landing loop. Big, isn't it?

1900-2050 Look at all the stuff we have to set up each time! Most of these variables are self-explanatory or nearly so. Especially if we tell you that "pos" means "position" and "vel" means "velocity". FUEL is actually fuel remaining, while BURN is the current rate of burn (thrust). BURN is the number which is adjusted by moving the joystick back and forth. CURRENTTHRUST matches BURN only if the button is pushed, otherwise it is zero.

2060-2140 We set up the fuel-remaining indicator. Rather than a solid bar, we liked the pattern that $BDDB produced for a pair of vertically adjacent lines within the bar. We replicate the pattern via the MOVE of line 2090. Note how this trick works and use it in your own programs: If you initialize the first N bytes of an area of memory, you can replicate those bytes via

```
MOVE area,area+N,(N of replicates/N)
```

Another trick you might steal is our method of moving character shapes from ROM to a player (lines 2100 to 2130). The usual character set starts at $E000, but we bias it by -$100 because screen byte values are no identical with ATASCII values. Recall that each character in ROM occupies 8 bytes, and you should get an idea how this works. After the "fuel line" is ready, we move it to the left side of the playfield screen.

2160-2510 We make the playfield look pretty. After picking the size and width of the landing pad, we draw the moonscape in three pieces: From the left edge to the pad (line 2290), the pad itself (2310 to 2340), and from the pad to the right edge (2360). The subroutine at line 3980 draws the jagged mountains. (Note how the mountains are guaranteed to get no more than 20 units high. If

ALT gets up to 2Ø, Ø.96*ALT immediately drops it back to 19. Cute.)

After putting a few distracting stars in the sky, we blink the landing pad (that's one reason it was drawn using a different COLOR than the rest of the moonscape) and then give it the same color as the rest of the mountains.

26ØØ,277Ø This WHILE loop constitutes all the actual movement in the game! Do you see how few lines there are here? That's the primary reason the game can run so fast, thanks to the extensive set up which we have done. And what terminates the movement loop? Look at the five conditions in the WHILE statement: (1) Hitting the landing pad. (2) Hitting the mountains. (3) Going off the left edge of the playing area. (4) Going off the right edge. (5) Going off the top of the area.

261Ø-262Ø We move both the lander and its thrust flame into position. For vertical movement, we actually MOVE data from the strings we set up (from the hex DATA). We do this because it is faster than PMMOVE, which must move 512 bytes in single line resolution (256 bytes out to a buffer and then back in, to avoid overlap problems). For horizontal movement, PMMOVE is just as fast as POKE, so we use it.

263Ø-273Ø After adjusting the BURN rate as requested, we set CURRENTTHRUST to either zero or BURN, depending on whether the button is being pushed. Since fuel is used at a rate equal to Ø.1 times the thrust, we use an intermediate variable (LOSS) to accumulate thrust in units of 1Ø. When the LOSS exceeds 1Ø, we use up a unit of fuel and reflect that fact in the fuel line on the left side (lines 271Ø to 273Ø).

274Ø-276Ø The horizontal velocity is easy: we just accumulate the horizontal stick pushes in one-twentieth of a unit increments. The vertical velocity is also cumulative, but It uses the elements of the THRUST array for its acceleration values. And, you may recall, the values in THRUST() depend on whether you are playing at beginner or advanced level. Finally, after updating the horizontal and vertical positions, we make an appropriate rocket sound.

28ØØ-3Ø6Ø For really great landings, we bring out the crowd. Note the way we assign the bonus points in line 3Ø6Ø.

3Ø7Ø-325Ø For so-so landings, we bounce the ship. The number of bounces depends on how hard the landing

was. Note how we choose the frequency for the plopping sound from the PLOP() array.

3270-3650 A crash landing. We allow pieces of the ship to spew all over the place. Up to 10 pieces are given independent positions--X() and Y()--and velocities--XVEL() and YVEL(). Each follows the laws of physics until it goes off the playing field.

3660-3740 We display the score for this landing as well as the cumulative score so far.

3770-3870 After five landings, we give the grand total. We restart the game (via a simple RUN) when the joystick button is pushed (which is why we waited for the button to be released up there at the beginning).

There it is. A practical real-time game written entirely in BASIC XL. There are a lot of unnecessary frills (e.g., the various types of landings), but they add to the overall effect of the game. Try this on your Apple-owning friends. They'll never believe it was done entirely in BASIC.

9.2.5 GTIATEST

The earliest Atari computers had a graphics chip called a CTIA. About two years after their introduction, though, Atari started shipping all 4ØØ and 8ØØ machines with a newer chip, called a GTIA. (All XL computers use the GTIA.) The most significant difference between the two chips is the GTIA's ability to accept commands for three additional graphics modes, GRAPHICS 9, 1Ø, and 11 in BASIC parlance.

For reasons we at OSS find hard to understand, little in the way of commercial software has been produced which uses these three modes. True, compatibility with older machines is an issue, but the cost of a CTIA to GTIA upgrade is nominal, at most. And if you must maintain compatibility, why not provide two versions of a program? Well, one argument for not doing so was that, according to Atari literature, there was no way for a running program to tell which chip was installed. Would you believe Atari literature?

We thought not. It turns out that a workable method is a bit involved but more than doable. The subroutine from line 9ØØØ up in this program demonstrates one way which we know works.

The principle is as follows: If you are in a text mode (e.g., GRAPHICS Ø) and you turn on one of the GTIA enable bits (the upper two bits of GPRIOR), then the collision detection mechanism does not work between a player and a character displayed in the modified text mode. As a sidelight, the characters become unreadable under these conditions, but this in itself is not detectable by a program.

We believe this subroutine (and its sample calling program) are fairly self-explanatory, but we will make a few comments.

91ØØ As long as we are testing, we might as well PRINT something which makes sense.

913Ø-915Ø All of this ensures that we will place a black bar (player Ø) right over the word GTIA.

916Ø-921Ø We turn on the GTIA bits, wait for a clock tick, clear the collision registers, then wait at least two clock ticks.

922Ø If $DØØ4 contains any non-zero bits, it means a collision was detected and that the machine under test does not have a GTIA.

We hope that some of our users, either of BASIC XL or other languages, will see fit to produce some programs which take advantage of GTIA graphic modes when possible.

9.2.6 CIRCLES

We at OSS cannot take credit for discovering the algorithm used in this program, but we do think that we have made it a little more useful.

The program's workings are certainly self-explanatory up to line 1590. It is the subroutine starting at line 1600, which actually draws the circles, which needs a few comments.

The principle involved is simple in theory: calculate the sine and cosine of angles which get increasingly larger (until they reach 45 degrees), and plot a circle by reflecting these values in all octants. The trouble is, if we use conventional means of generating sine and cosine values, drawing a circle takes so long we might want to take a nap. The trick here is an algorithm, involving the variable DELTA which approximates the sine and cosine values so close as to be indistinguishable when a circle is plotted on an Atari-size screen.

When we enter the subroutine, we assume that XC, YC, and RADIUS are already set up. Then comes the fun.

1670 This begins the real work. The formula for DELTA is magic. Don't question it (unless your math is a whole lot better than average). The values for X and Y are more obvious: We begin at an angle of zero degrees, so the sine is zero and the cosine is one. We will plot the points where lines parallel to the axes intersect the circle.

1680 This allows us to get to 45 degrees, where the sine and cosine values are identical.

1690-1780 We plot the values in all octants. The cute trick we added here was the TRAP statements. Even if the circle is completely outside the bounds of the playfield, we can PLOT it in theory at least! The beauty of this method is that all of those points which fall within the playfield will be plotted, no matter how few or how many they are.

1800-1840 This is the algorithm at work. Again, It's partly magic, but you can sort of see how it works. X is always increased by one, so we never plot the same point twice. Whether or not Y is decreased by one depends on the value of DELTA (which in turn depends on either X or the difference between X and Y) as its sign changes. Those of you with a mathematical streak may enjoy calculating the arc-tangent of X/Y, to see how close this algorithm is.

Once again, this subroutine is one you can use in your own program. Try it, it works.

9.2.7 DISKIO

This is another program which in and of itself is only marginally useful. Its main purpose is to present its primary subroutine (lines 9000 and greater), which you may use in your own programs.

As you may or not be aware, when you ask BASIC to do I/O (Input/Output) to or from most devices attached to your computer (including particularly the disk drive), what actually happens is quite complex. BASIC Interprets your request, into a call to CIO (Central Input Output), which in turn determines what device you are using and vectors to the appropriate driver routine. We assume here that CIO accesses FMS, the File Management System for the disk, usually called DOS (Disk Operating System).

Finally, FMS makes a call to SIO (Serial Input Output), the routine which does the actual physical reading and writing to the device. In the case of the disk drive, this involves the actual transfer of a single sector of 128 bytes (or 256 bytes in non-1050 double density).

Most BASIC programmers seldom -- if ever -- have need to read or write a physical disk sector. Writing is dangerous, since disturbing the format of portions of a sector can destroy DOS's ability to manage the disk for you. Reading a sector, though, can be informative, especially if you are trying to either understand DOS or find "lost" information.

However, should you ever feel the need to directly read or write sectors, the subroutine we provide here will do the work for you. Just so you can see how it works, we have included an interactive program which reads selected sectors. (We took our own advice and didn't allow it to write sectors.)

The set-up program, all lines except the subroutine startling at line 9000, is fairly self-explanatory. It simply asks the needed questions before calling the actual read-a-sector code. It then displays the contents of the sector in an easy to read hex and ATASCII dump format. Only a couple of points are worth making regarding this part.

First, we have arbitrarily used $600 through $6FF as our sector buffer. This is the infamous "page 6" which is so often overused. If you would like to avoid conflicts with other routines using page 6, feel free to locate the buffer anywhere else (e.g., within a DIMensioned string). Second, note the way we print out the dumps. The HEX$() function always returns a four-character string; but, because we want only the last

two (least significant) digits, we assign its value to a temporary string from whence we can print out only the last two characters. Also, we avoid problems with the ATASCII display by prefacing every character with the ATASCII code for ESCape and ensuring that only seven bits of the characters value are used in the display. The former mechanism forces E: (the screen device here) to display what would otherwise be cursor control codes, etc. The latter "fix" ensures that RETURN ($9B) won't be sent to the screen, a desirable feat since it overrides even the ESCape sequence.

And now, before describing the code in the sector access routine, we need to examine what SIO expects to be where when it is called.

9.2.7.1 SIO and the Device Control Block

The entry point to the SIO calling routine is located at $E459. When SIO is called, it does not care what values are in the various CPU registers (A,X, and Y), but it insists that a block of memory known as the Device Control Block (DCB) be properly set up. There is only one DCB used in the Atari OS, and it begins at location $Ø3ØØ (768 decimal). Its contents are as follows:

Location	# of bytes	Description
$Ø3ØØ	1	Physical Device ID
$Ø3Ø1	1	Device Unit Number
$Ø3Ø2	1	Device Command Character
$Ø3Ø3	1	Data movement control (on call) SIO Returns Status (on exit)
$Ø3Ø4	2	Buffer Address
$Ø3Ø6	2	Timeout value
$Ø3Ø8	2	Buffer Length
$Ø3ØA	2	Auxiliary Information

Some of those brief descriptions need a little explanation: The physical device ID is something not seen in Atari's OS outside of SIO. Atari has assigned each standard serial peripheral type a unique ID; disk drives have an ID of $31 ('1', not to be confused with $Ø1). The device unit number is more familiar as, for example, the drive number ('n' in 'Dn:').

The device command is again unique to SIO. As we shall see in the next section of this manual, there are many possible command characters, though they tend to be normal ATASCII letters. For example, the command to read a sector is 'R' while write is 'W', Note that for versatility disk drives support a second write command, 'P', which means write sector without verify.

The byte at $3Ø3 has two uses. When you call SIO, it must contain $4Ø if you wish to obtain data from a device or $8Ø if you need to send data. A few device control commands need to neither read nor write data, so they use a value of $ØØ here. On return from SIO, the error code value (if any) is placed in this location.

Buffer address and buffer length are similar, if not identical, to their CIO counterparts. They simply tell SIO where the data is and how much of it there is. One unfortunate point: ATARI did not choose to include the data length in the packet sent out over the serial bus. This means that the device and SIO must agree on the length of data being sent. (Example of the consequence: Atari's OS always sends data to a printer in 4Ø byte chunks. Wouldn't it have been simpler if OS could have sent any number of bytes, from 1 to say 255, to the printers?)

Finally, the auxiliary information is sent unmodified to the device along with the command. Each device chooses what the auxiliary info implies, but for disk drives it is always the sector number.

9.2.7.2 The Sector Access Routine

Actually, now that you have seen what SIO requires, this subroutine (lines 9ØØØ up) is almost self-explanatory. Once again, though, a few things need clarifying.

923Ø No real reason for this, except that the resultant listing looks so much neater.

924Ø We use ASC("1") to emphasize the fact that Atari, for some reason, used printable characters for most of the SIO control information. (As a guess, we would say that they did this to make debugging using a serial data analyzer easier.)

927Ø-932Ø We only allow the values we said we would. Everything else is fatal. Not fancy, but safe.

933Ø A little sneaky, but we have already verified that CMD equals either 1 or 2, so only a legal value is possible here.

935Ø The timeout value is arbitrarily large.

936Ø-941Ø Again, we allow only legal density values. Note that 1Ø5Ø density-and-a-half is considered Single density by this routine.

9420-9470 Validating the sector number. If you are using a 1050 in density-and-a-half mode, you obviously need change the 720 value to 1040, instead.

9480-9490 This is such a neat trick! Because BASIC XL allows us to specify that the count of parameters will not be pushed on the stack, we can call machine language routines which do not expect values in registers without any need for an intermediate routine! So simple it's almost hard to believe.

9500 As advertised.

9510 Just in case the caller is using a routine where he wants the count of parameters pushed!

9.2.7.3 Technical Sidelight

There are two sectors on a standard Atari DOS disk (version 2.0s and its derivatives, including OS/A+ and DOS XL versions 2) which you may read or write at will, since they are "invisible" to DOS: sector 3 and sector 720.

Sector 720's availability has been documented before: DOS "manages"·sector numbers 0 to 719, but the disk drive understands only sectors 1 to 720. DOS has been "fixed" to think that sector 0 is always in use, but sector 720 remains outside its Ken. Sector 3 is a quirk: it is the last sector of the traditional 3-sector boot process. But, for some reason lost in programming legend, it turns out that none of the disk boot code used by DOS is present in sector 3: sectors 1 and 2 contain all the boot that is needed!

A word of warning, though: if you erase, write, modify, or rename the DOS.SYS file, sector 3 will automatically be rewritten by DOS (it thinks it needs to reestablish the boot code). So, if you choose to use sector 3 for your own purposes, be sure to do so on a disk which either never receives a DOS.SYS file or which has one which you feel is reasonably permanent.

9.2.8 CONFIG

This program was written in response to all of our users who wanted to know how to read and/or change the configuration information which all true double density drives utilize. The configuration scheme, often called the config block, was developed by Percom Data Corporation, the producers of the first commercially available double density disk drive for Atari computers. Since that time, all other manufacturers except Atari have followed the Percom lead. Strangely enough, the Percom scheme was in turn developed from the ill-fated Atari 815, a double density drive which never saw retailers' shelves.

In any case, the degree of double density compatibility between drives of rival manufacturers in the Atari market is nothing short of amazing. In those instances where one drive cannot read a diskette written by another make of drive, the problem is almost always related to the rotational speed of the motor turning the disk. Adjusting that speed can often work wonders with a diskette which otherwise produces only ERROR144.

Of course, when Atari finally came out with their own "double density" drive, naturally they had to invent a new standard. (It wouldn't do to accept one begun by a rival--that would be an insult to Atari's dignity.) As a result we now have three important diskette configurations in the Atari world, which are summarized in the chart below.

Our Name	Style	Sectors per Track	Bytes per Sector	Kbytes
Single Density	810	18	128	90
1050 Density	1050	26	128	130
Double Density	Percom	18	256	180

All drives use 40 tracks per diskette. In addition to those shown, various manufacturers have also made drives with 80 tracks, two heads (i.e., 40 tracks per side of the disk), double-headed with 80 tracks per side, and even 8" disks with other strange and wondrous configurations. Since only OS/A+ version 4 (of all OSS DOS's) supports other than ordinary single and double density drives, we will not go into detail about these drives here.

As of this writing, the following drives are known to be capable of understanding Percom-standard double density mode:

Indus	TRAK
Astra	Rana

SWP NCT Turbo
and, of course, Percom

In addition, Amdek conforms to the software standard even though their diskettes are 3.5" (instead of the usual 5.25"). If you hook a 5.25" drive up to an Amdek controller (e.g., as a second or third drive on the controller), then its diskettes will be hardware compatible as well.

Now that we have all that out of the way, maybe we ought to find out just what the "Percom standard" is.

9.2.8.1 The Percom Standard

For a drive to qualify for that title, we at OSS feel that it must be capable of all the following:

1. Read and write standard Atari 810 single density diskettes.

2. Read and write double density diskettes with 40 tracks, 18 sectors per track, 256 bytes per sector. Peculiarity: because of the way Atari's OS wants to boot, the first three sectors of a double density disk will hold only 128 bytes of data (excess is ignored) and transfer only those 128 bytes on all SIO reads and writes to sectors 1 through 3.

3. Be able to transfer an internal configuration block to the host computer on request.

4. Be able to accept changes in that same configuration block sufficient to at least allow the drive to be changed back and forth between single and double density.

5. Have that configuration block be read/written by SIO commands 'N' and 'O' (respectively) and consist of 12 bytes conforming to the following table:

Byte #	# of Bytes	Description
0	1	Number of Tracks
1	1	Step Rate
2	2	# of Sectors per Track
4	1	# of Sides per Track
5	1	Density (0=Single, 4=Double)
6	2	# of Bytes per Sector
8	1	Drive Selected?
9	1	Serial Rate Value
10	2	Miscellaneous (reserved)

Once again, a little explanation of some of those items is necessary: First of all, note that all double byte values are not in standard 65Ø2 low/high order. The reason is historical: Percom uses a 68Øx CPU chip in their disk controller, and all 68Øx chips do double byte work in reverse of the 65Ø2 manner.

'Step Rate' is not a meaningful number from one manufacturer to another. Step rate 1 might mean 6 milliseconds per track to one manufacturer and 2Ø milliseconds each to another.

"Number of sides" is a misnomer: it is actually the number of sides minus one. Thus most drives will show a zero here. Note that, in theory, this number could have any value. For example, a hard disk drive might show 4 here (five heads).

The only agreed upon values for "Density" are Ø ("FM" recording mode) and 4 ("MFM") recording mode. Other values are possible for strange circumstances.

Some drives can actually be turned "off-line" by an appropriate value in "Drive selected." There seems little value in this, since they can only be brought back into the system by turning them off and back on again.

The "Serial Rate Value" has not found any compatible acceptance. As originally conceived by Percom, it would inform the drive what baud rate the computer would use for high speed data transfer. So far, those manufacturers offering higher speed transfers have not used this byte in any meaningful way.

Finally, the "Miscellaneous"· value is not—to the best of our knowledge--being used by anyone for any purpose.

Now that you know what a Config Block looks like, how can you tell, from software running in the Atari computer, whether a particular disk drive is set up for a particular density of diskette? Equally important, how can you change a drive's set up? If you want the answers to these questions, read on.

9.2.8.2 Reading and Writing the Config Block

As noted in section 9.2.7, SIO is a means of transferring control and/or data between an Atari computer and a peripheral device via the standard serial bus. Although the most common operations on the bus involve reading (command 'R') and writing (commands 'W' or 'P'), other commands are certainly possible. In fact, all devices are required to support a status ('S') command, if for no other reason than so that the

computer can tell whether they exist on a given bus or not.

When Percom invented their double density disk drive, they invented their Config Block and, quite naturally, a pair of commands to pass such a block between the computer and the drive.

The command to read a Config Block from the drive into the computer memory is 'N' (think of it as iNto the computer). The command to write a Config Block to a drive is 'O' (think of it as Out of the computer). Aside from the need to use these command characters, the only differences between making an SIO call to read/write a sector and making one to read/write a Config Block are (1) the length of the data, which is always 12 bytes (instead of the 128 or 256 for a sector) and (2) the auxiliary bytes (used for sector number) have no effect.

For example, then, to read a configuration block from drive 1 into a buffer at location $600 (page 6) you would need to set up the following values in the DCB at the locations shown:

```
     $300  $31   Unit ID
     $301  $01   Drive 1
     $302  $4E   'N', read Config Block
     $303  $40   see section 9.2.7
     $304  $00
           $06   $600, LSB first, buffer address
     $306  $0F
           $00   15, an arbitrary timeout value
     S308  $0C
           $00   12, length of the Config Block
```

And that's it! A JSR (or USR) to location $E459 will read that block right into memory. If, of course, the drive is capable of reading/writing Config Blocks. Atari drives, for example, will return an error 138 (NAK), because they do not understand the command. A command given to a drive not on the serial bus will result in a time-out error.

1000-1200 Mostly just simple constants. Note that we will read the configuration table into the string, Config tables, rather than using valuable page six memory. Also note in line 1200 the way we produce screen control characters which will list on any printer.

1240 This allows us to call system routines via USR() directly. See section 9.2.7.

1270-1290 We will discuss these DATA statements later. For now, note that each line has 12 values (funny how that matches the size of a Config Block). Negative values indicate bytes we won't change.

1330, 1920 Look at the size of this endless loop. We think that, in a well structured program, a loop really shouldn't get any bigger.

1340 Two ways to use screen controls in BASIC XL, thanks to the fact that you can PUT to channel zero.

1430 This is one way to ensure that all the configuration games we are playing here will take effect. When you change a drive's configuration, DOS needs to know about it. Usually, one does this by calling a routine named DOSINI, which will return to you after reestablishing DOS's internal drive configuration table. If you don't need the routine to return to you, simply force a system reset by a jump (of any kind) to $E474. This is exactly equivalent to hitting the RESET key.

1490-1500 See, we can use our SIO calling routine to do more than just read/write Config Blocks. In this case, we simply do a drive status call.

1600-1730 The status was okay, so read the Config Block. Hmm? Can't do it? Why did you buy an Atari drive?

1750-1890 Here is where we display and then (optionally) change the Config Block in a form readable by humans. Note how little of the code is actually here; it is almost all in subroutines.

1940-2220 Once again, we have a keyboard access routine which avoids the vagaries of the INPUT statement (see PICO.BXL for a fully commented example of this same thing). In this case, we want only numbers in the proper range. It's easy if you step through it.

2230-2590 Remember what we said about a handful of subroutines which do the real work? Here's one of them. If you followed our discussion of the meaning of each byte of the configuration table (above) you shouldn't have any trouble following this code. That's primarily thanks to the fact that all the pertinent values have already been placed in variables with meaningful names by...

2600-2750 A very important subroutine. This takes the bytes of the Config Block and converts them as appropriate. Note how we can not use the DPEEK() function, thanks to the fact that the double byte

values are "backwards"·compared to standard 65Ø2 practice.

2760-2910 The opposite of the previous routine. Take the values in the variables and stuff them into the bytes of the Config Block. Again, note that we can not use DPOKE.

2920-3120 We really shouldn't need to explain this routine, since it is virtually identical to its counterpart in DISKIO, described in section 9.2.7.

3130-3380 Here's where we allow you to play games, if you wish. We give you a menu. If you choose one of the standard configurations (Single, 1Ø5Ø, or Double Density), then the appropriate RESTORE allows us to read the standard configuration information from our DATA statements. Once again we note that some bytes are never changed: Step Rate, Acia, and the Miscellany locations.

3390-3900 Anything goes. You can tell the disk drive's controller that it's connected to a drive with 13Ø tracks, 2Ø4 bytes per sector, 12 heads, or whatever. Some controllers will believe you and try to do as you ask. We sincerely hope that you have a blank or trash diskette in the drive when you give such commands. Other drives will only accept a limited number of configurations, ignoring much of the information you send them. For example, Indus drives allow only the three standard densities.

Note how we re-read the Config Block after writing. This is to ensure that we haven't lost control of the drive. (With same drives, you can de-select them, and they will cease responding to anything.)

That's about it. If you are confused, try playing with the program with a copy of a listing in front of you. It should become a bit clearer.

9.2.9 PHONE

PHONE.BXL is a fairly large but well organized program which is a simple but very efficient phone number list organizer. It will maintain a list of first and last names and phone numbers, keeping the list "sorted" by last name. Thanks to the "sort" scheme adopted, it finds a phone number in less than a second, no matter how many names there are in the list, when given a last name to work with.

Its other advantage is that it is easily changed and expanded to provide, for example, a mailing list program. Or perhaps a list of books in your library. The possibilities are limited mostly by your willingness to tackle its code and bend it to your purposes.

Again, this program has been provided in response to numerous requests for a complete explanation of how to do random access file I/O under DOS 2. We hope that this program and its description will satisfy most of these requests. Before exploring the program, though, there are several technical considerations which you may enjoy considering. If you get lost in all the technical stuff, skip down to the program description and come back and try to understand the rest later. (It is worth understanding.)

9.2.9.1 Sequential and Other Files

Perhaps the biggest flaw in Atari DOS 2.Øs (and all its derivatives, including OS/A+ and DOS XL version 2.x) is in the structure of the files it creates. Atari DOS 2 files are classified as "linked sequential" types. That means, each sector in the file points to (links to) the next sector.

Sequential files have a few advantages: (1) File managers which handle sequential files are generally simpler and smaller than those for other file types. (2) If a disk is partially "clobbered," you can often still recover much of its data when linked sequential files are used. This is true even if the disk's directory is damaged,a generally fatal condition in other file systems. (3) File manager disk space overhead is reasonably low.

Unfortunately, there are also several major disadvantages: (1) To erase a linked sequential file, the file manager must read through each sector of the file, a very time-consuming process. As disk and file sizes get larger, this become a major factor in disk I/O time. (2) To locate a particular record in a linked sequential file, you generally have no choice but to

start at the beginning of the file and read until you come to it. (3) Similarly, to append to a linked sequential file, you may have to read the entire file.

Now, truthfully, file manager types don't matter if you are using a DOS to do nothing but save programs, letters, and other things where you always load all the information into memory before working on it. You're actually using the disk as a slightly smart tape drive in these circumstances. Where file structure be comes important is when you need to randomly use bits and pieces of a hunk of data (a file) too big to fit in memory.

The best of all worlds would be a DOS smart enough that you could say something like this: "Give me the address of John Doe." Generally, the computer world considers convenience like this beyond the scope of DOS, relegating it to the world of Data Base Managers and their ilk.

The next step down is usually being able to say, "Give me the 433rd record in that file." With most file organization schemes, this is a trivial task if the records are all the same length (and about as hard as the first request if they are not).

9.2.9.2 How to Use NOTE and POINT to Advantage

But what about those linked sequential files we are stuck with? To get to the 433rd record, we have to read through the first 432! And we would be stuck here were it not for the fact that Atari DOS does provide one added feature: it allows you to find out just where on the disk you are as you read or write a file. The magic statement is NOTE. As you may remember from your BASIC XL reference manual, its format is:

```
NOTE # filenumber,avar1,avar2
```

where the first avar gets the sector number of the current position within the file and the second avar gets the byte number within that sector.

Then, if you once read a file and find out (via NOTE) where its 433rd record begins, you can later ask DOS to change its file position marker to that same location (via POINT, which has the same format as NOTE). Voila, you are then able to read or re-write the record.

How, you may wonder, is this different from those DOS systems which allow you direct access to any byte (and thus record) in a file? Don't they allow you to POINT to any disk location, also? Not really. Atari DOS allows only what we call Absolute access. That means

that the numbers you use with POINT describe a physical location on the diskette. Other DOS types allow you to POINT to a location which is relative to the beginning of the file. (Example: To point to the 22nd record when each record has 20 bytes, you would simply POINT to relative byte number 440, if records are numbered starting at zero.)

With Atari DOS, knowing that record number 22 starts at sector 301, byte 115, doesn't tell you anything about where record number 23 starts (unless record 22 is shorter than 10 bytes), because sectors are not always allocated to a file in order. (Instead, as a file is built it is always given the next unused sector.) To make matters worse, when a file is appended to, sectors with fewer than 125 bytes (253 bytes in double density) may be left in it.

The only real solution, then, is to build a table of pointers, one per record. This technique has been described often before (among other places, in Atari's DOS 2.0s Reference Manual). In most such discussions, what is built is a numeric array (or arrays) of pointers to records by number. A segment of a typical program is shown:

```
950 NOTE #3,Sector,Byte
960 Sector(Recordnumber)=Sector
970 Byte(Recordnumber)=Byte
```

This is a lot of overhead: 12 bytes per record.

Let us sidetrack for a moment. Consider this: when you use NOTE, you are given a sector number and a byte number. But the maximum sector number is 720 and the maximum byte number is 253 (double density), so we can store the sector number in as little as two bytes (remember, a double byte location can hold values from 0 to 65535) and the byte number in a single byte. Total: three bytes. Again, a program fragment to implement this scheme is shown here:

```
930 NOTE #3,Sector,Byte
940 Temp=Recordnumber*3+1
950 Shi=Int(Sector/256) : Slow=Sector&255
960 Pointer$(Temp,Temp+2)=Chr$(Slow),Chr$(Shi),Chr$(Byte)
```

Look at the savings when compared to the numeric arrays! But an additional advantage of using a string to hold our pointers is that it can hold any other string as well. Why not a record's 'name'?

If you are using 100 byte records, a file with 500 records needs only 1500 bytes worth of pointers, which can easily be held in memory. Even if you add "record

names" (as PHONE.BXL does), the memory requirement for a set of pointers is quite small compared to the amount of disk space we can access with them.

And, while you could re-build the pointers each time you RUN a program, isn't it just as easy to keep them in another file on the disk? Yes! And all of this is made so much easier thanks to same statements in BASIC XL. There is, however, a necessary caveat: Recall that the sector and byte numbers given you by NOTE are absolute. If you copy the data file to another disk, your set of pointers is no longer valid. You thus have two choices: rebuild the pointers after copying the data file or duplicate the entire disk instead (which preserves everything on the disk).

9.2.9.3 The Concept Behind PHONE.BXL. alias BlackBook

It's kind of funny that, because other COS systems support random access files implicitly, you seldom see programs such as this published for them. And what's so special about this program? In it we give you a complete set of routines for performing what is known as an "Indexed" or "Keyed Sequential Access Method". Remember how we said it would be neat to be able to access John Doe's account information using just his name? Remember how we said this was in the domain of Data Base systems? Guess what. PHONE.BXL (or, as we prefer, "BlackBook") is actually a mini-Data Base. All in all, we have turned a DOS limitation into a helpful situation.

(Sidelight: Actually, there is no reason you couldn't use all the techniques of this program under any DOS. In fact, most random access would make some of the steps in our process -- such as "prebuilding" all data files – unnecessary.)

BlackBook always works with its files in pairs: a data file and an index file. The structures of the files are shown below:

9.2.9.4 BlackBook Data Files

Each record consists of three fields. Each field is a string of up to 24 characters which is written to the file via BASIC XL's RPUT statement. Since RPUT uses five bytes of overhead per string (as a safety measure -- see your reference manual), the total number of bytes per record is 87 (24+5 is 29; 3 times 29 is 87). If you were to look at a record byte by byte, it would look like this:

Record Structure in BlackBook Data File

```
Record:
     Field 1:
          Byte  1      String Field indicator
          Bytes 2-3    Dimension of String Field
          Bytes 4-5    Length of String Field
          Bytes 6-29   Field data, as a string
     Field 2:
          Byte  30     String Field indicator
          Bytes 31-32  Dimension of String Field
          Bytes 33-34  Length of String Field
          Bytes 35-58  Field data, as a string
     Field 3:
          Byte  59     String Field indicator
          Bytes 60-61  Dimension of String Field
          Bytes 62-63  Length of String Field
          Bytes 64-87  Field data, as a string
```

9.2.9.5 BlackBook Index Files

Aside from the actual key (index) entries, there are two pieces of information needed when maintaining a keyed file as BlackBook does: (1) We must know how many records the file is capable of holding. This number -- called MAXREC -- is established when the empty file is pre-built. (2) Out of those MAXREC records, how many are currently in use? NUMREC tells us.

In BlackBook, MAXREC and NUMREC are placed first in the index file via RPUT. They are directly followed by all the bytes of the index string. Since MAXREC describes the size of this string, we chose to write/read it with BPUT. (There is another advantage to using here, as we shall see later.) The byte-by-byte form of an index is thus as follows:

```
     Byte  1     Numeric field indicator
     Bytes 2-7   MAXREC, a number
     Byte  8     Numeric field indicator
     Bytes 9-14  NUMREC, a number
     Bytes 15-?  The index string
```

Sidelight: the reasons we set up the files for MAXREC records, instead of just adding space to the file as we need it, are twofold and related: (1) You can only use POINT on a file which has been OPENed in update mode. (2) You can't append to a file when you are in update mode.

9.2.9.6 The Index String

The proper structure to the index string is the secret to not only the success but also the speed of this program. Rather than trying to explain it as we describe the workings of the program, we will present it in some detail here.

The string actually consists of MAXREC "elements", just as if it were an array. In BlackBook, we have chosen to use the first four characters of each person's last name as our key value. This is arbitrary and could, without a lot of trouble, be changed. (In fact its size is dependent on the value of the Indexsize variable.)

In addition to the 4 character key, there are 4 bytes of overhead. Three of them we know about: two bytes for the sector number, one byte for the byte number. The last byte is used as a key separator and always has a value of 255 ($FF). At this point, you may be wondering why we went to the trouble of using a long string (with its complicated subfield addressing) in favor of a string array (where we could get the entire key pertinent to a record with a simple record number). One main reason: BASIC XL's FIND() function works only on a single string (not an array), and we wanted to use it for speed.

But using FIND() has its own problem. Suppose that, just by coincidence, the sector and byte number characters (which is what they have become, once they are in the string) happen to have values which make them look like characters in a key name we are searching for with FIND(), causing the function to return a false match. We avoid the problem through the mechanism of the $FF byte field separators: When we search for a key name with FIND(), the search string is preceded by a byte of $FF. A match is thus guaranteed to start on a key separator boundary. (We go further for safety: we separate the sector number into high and low bytes by dividing by 128, instead of the more conventional 256. This means that the sector number and byte number characters can never have a value of 255 either. Overkill? Perhaps, but why not when it costs us nothing.)

Got all that? If not, don't worry about it. If the description of the program still doesn't make it clear, it doesn't matter. If you follow our lead, the scheme will always work.

9.2.9.7 Program Description: PHONE.BXL, BlackBook

If you list this program to a printer (and we sincerely hope you do before trying to follow this description), you will find that it will take over 8 pages of paper. Obviously, there is no way we can give you a line-by-line description of such a program. Instead, we can only point out the functions of various subroutines, etc. Of necessity, some of the detail about program function, etc., given in other descriptions will be missing here. We hope and expect that your programming skills will have been sharpened enough by now to allow you to work through the details.

As with some of our other programs, we will describe this program from the "top down". That is, we will present it in roughly execution order rather than listing order.

1000-1330 The usual constants, both strings and numbers. Note how we have given "names" to commonly used small numbers such as zero and one. This saves memory space, not time.

1340-1430 If you adapt BlackBook to your own purposes, you can add data fields here and/or change the sizes of the ones given. If you do so, be sure to adjust Recsize, the number of bytes in each record. If you choose to change the length of the portion of a field used as the key, change Indexsize at your own risk. In theory, everything in the program keys off this variable, but we have never tested the theory.

1540-1550 One advantage of using BGET with the index string is that we do not need to make the DIMension of Index$ match the size in the file, as we would if we used RGET. This makes building a file somewhat easier also, as we shall see.

1570-1710 We have given all major subroutines names in this program. This makes renumbering and reorganizing a bit more difficult, but pays off in much more readable code.

1720-1950 Did we mention that BlackBook will even dial your phone for you? Here, we're just setting up an array of values for later use with SOUND.

2000-2290 This monstrous program is all driven from these few lines. All we do is present a menu and accept only one of five choices. If you are using BlackBook, you have to create a file before you can do anything else, so we will now track what happens when you ask for that main menu option.

15ØØØ-1528Ø This major routine figures out how big a file you have, allows you to specify any size up to that maximum, makes you choose a name for the file, and creates an empty data file corresponding index. Such a lot of work for so little code! It's done with mirrors, otherwise called subroutines.

69ØØ-7Ø6Ø The Calcsize routine. It figures out how big a data file is possible using a trick or two we hadn't seen before. First, it creates a trash file containing 2ØØ bytes. It does this so that it can read the sector count for this file in the directory: 2ØØ bytes is guaranteed to require one sector in double density, two sectors in single density. Then, when it finds out how many free sectors there are, it knows how many free bytes there are on the disk. From this count of free bytes it estimates the maximum number of records by dividing by the number of bytes used by each record, which is in turn the sum of the record size and the index size. Finally, we never allow ourselves more records than we have room to point to in the index string.

5ØØØ-526Ø Our Getline routine is used to avoid the INPUT statement. We avoid INPUT because we don't want the user moving the cursor all over the screen, erasing the screen, etc. Either the ESCape key or the RETURN key terminate a line here (mainly because they have the same value if you ignore the upper bit). The only editing key we allow is Back Space, and then only to the beginning of the field. We even provide for the use of a flag which changes lower case into upper case, used by the Getfilenames routine to avoid lower case in file names. Finally, we will only get as many characters as the caller asks for (the contents of Maxline on entry).

54ØØ-548Ø Getfilenames is only a little bit smart. The user should not type the file name extension, and typing the drive specifier is optional (D1: is provided automatically if the specifier is omitted). Two names are returned, alike except for the extensions, DBF and DBX (Data Base File and indeX).

75ØØ-768Ø Most of the work in Create is done by this routine, Makeindex. Since this is a very important routine, we will examine it in some detail. Exception: As this routine works, it keeps the user informed of where it is. The code for this is fairly obvious and will not be discussed.

We first set up the data fields with some filler byte ($FF, in fact). After performing a NOTE (line 757Ø) to find out where the beginning of the

current record is, we write the filler data to the data file (line 7610). As we did that, we built a key string. Note its structure (line 7590): first byte is always $FF (255), followed by four bytes which match the first four characters in the last name of the person being indexed, followed by the NOTEd information. We lengthen the index string (line 7600) by simply tacking the key we built onto the end of it.

We perform all those steps for each record in the file (the FOR loop). When all data records have been written out, we write out the new index file (lines 7640-7660). Note the presence of the check in line 7630: if the length of the index string doesn't correspond to the number of blank records which were set up, something went disastrously wrong. When writing your own code, checks like this are a good idea (but see our final comments also).

After creating a blank BlackBook file, you would presumably want to put some data in it. In this program, one main routine is used for operations on the data in the file: the Edit operations start at line 10000.

10000-10310 Once again, a major routine devolves to a small loop with many subroutine calls. And once again its primary purpose is to present you with a menu of selections and make you choose one. In the case of Edit, it first asks you a question and does a little set up.

7200-7320 Even though BlackBook files on only the first drive are listed for you, the Showfiles routine will accept a choice of a pair of files from any on-line drive.

7400-7460 Getindexinfo is a simple routine: it opens the index file, reads the count of available and in-use records, and gets the index string in place.

5700-5850 By never using zero as a real record number, we make Showrec's Job easy: If it sees us trying to display record number zero, it displays blanks instead. Note that the record number referred to is actually an 8 byte key entry in the index string, which may bear no relationship to the record's position within the data file. If you modify BlackBook to add fields, this routine must change to fit; but the POSITION and PRINT statements are easy to modify. To get the data to be displayed, this routine in turn calls...

6300-6340 Getbykey simply gets the various fields of the data record after requesting a POINT to the right spot in the file. Again, you could add data fields in each record quite easily in this routine, simply by extending the RGET statement.

6000-6070 Even deeper in the GOSUB information about sector and proper spot in the data file.

10240-10290 Finally, back in the Edit menu, we demonstrate a neat way of making menu choices using the FIND() function. The nice part about it is that an invalid choice provides an Option value of zero. Valid choices are vectored to the appropriate routine. For the sub-commands of Edit, we chose to use line numbers, primarily so we could renumber this section of the program more easily. Let's look at some of those choices in a logical order.

11110-11290 Again, on the assumption that we are setting up a new BlackBook file, we start by adding records. Since the Edit menu routine at lines 6600 through 6770 simply sets up a set of blank fields to be filled in, we won't describe it further here. The Getline routine does yeoman duty again, ensuring that we get nice neat data, confined to the proper areas of the screen.

Before bumping the count of records (as well as the current record number), we call two routines which do the bulk of our work. Observe how, in line 11250, we built up KEY$. By now, you know that an index string entry consists of a separator byte, four bytes of the record's name, and three bytes of NOTE info. But look where that NOTE info comes from here: from the last possible index entry in the index string! As you follow the next subroutine, you will see why.

7800-7930 This is potentially the slowest part of BlackBook when you are adding to a large file. Using a FOR loop, we search through the index string looking for a record whose name is equal to or greater than the one in KEY$. Because we never try to insert into a full index, we are guaranteed to find one such name: blank records were given a name of all $FF characters!

When we find the proper position to insert our new entry, we must make room. We leave it to you to work out how beautifully the MOVE of line 7900 works (though we will remind you that a negative length forces an insertion-type move). The special case shown is only used if we are putting the last possible name in and it happens, to fall at the end of the list.

Do you see what we have done? If this was the first real name being inserted into all the dummy names in the index string, its 8 bytes find their way to the beginning of the string. But look what data record we will use: the last possible one. So what? That's why we are using an indexed file, right?

6360-6400 Speaking of which, we new need to Putbykey to get the data record on the disk. As with GetbyKey, we let the PointbyKey routine set up the POINT for us and then we simply RPUT the data fields to the disk. It would be easy to add more data fields here, to correspond to GetbyKey.

Back in the Edit menu: Once you have added some records, you may want to go forward or backward in the file looking at what you have done. Or maybe you want to find a particular name.

10330-10450 As long as we're still within the bounds of valid data, we let the user go to the Next or Last (previous) name (alphabetically) in the file. Simple, isn't it? Thanks to the fact that the index string is already sorted in alphabetical order. (Well, that's really ATASCII order, but for names the difference is moot, unless some use upper case and some use lower case.) Notice that these routines do not need to display any data, since the main Edit menu loop does that for them.

10470-10590 This is why we went to all the trouble to set up that monstrous index string! See how we build our search name in line 10540, with a leading $FF byte. Then all the work is done for us in line 10550: we simply FIND the first match! Very fast, very efficient. Again, by calculating REC as a function of the position we found the name in the index string, we can let the Edit menu loop display the data for us.

And the only other things this program allows you to do with your data is dial a phone number or erase a name from the list.

10600-10950 This only works on touch-tone phone systems, but it does work. If you hold your phone's microphone up to your computer's speaker it is actually possible to let the computer dial for you. Some other things to note: A 'P' in a phone number indicates a short pause (some long distance companies need such pauses during dialing). You may easily adjust the duration of the pause by changing line 10780. A 'W' causes the dialer to wait until you give it the go-ahead. Once again, our friend the FIND() function passes through only those values we actually want to handle.

The tone generator uses the special 16-bit resolution mode of the Atari sound generators to produce frequencies which are more accurate in pitch than those available with the SOUND statement. The subject is too complex for further explanation here. Many graphic and sound books for the Atari explore this fairly fully.

1Ø96Ø-11Ø9Ø In most ways, the Erase a Record routine is simply the reverse of the ADD routine. We first remove the record pointer from the index string by simply squeezing up the string (lines 11Ø2Ø and 11Ø3Ø). But, because we don't want to lose the NOTE information in that pointer, we fill it in with the standard dummy name (all $FF characters) and tack it onto the end of the index string (line 11Ø5Ø). We mark the record as deleted in the data file by zapping just the last character of its PHONE$ string (11Ø6Ø and 11Ø7Ø). Naturally, the number of records is now one less than it was before.

Aside from the various edit options, the Edit menu provides an exit choice and a hidden choice (note the presence of the underline character in line 1Ø24Ø).

113ØØ-1137Ø To exit from the Edit menu, we simply close the data file and write out a new version of the index file. The next time we get to the Edit menu, reading the index file will put us right back where we left off.

99ØØ-994Ø In the process of developing this program, we had several occasions to doubt our sanity. Loops would straighten out. GOTOs wouldn't. Data would be lost. And the index string would get mangled unmercifully. To help view what was going on, we would often write small routines to display certain pieces of data. For example, we built in this debug routine, which simply displays the current contents of the index string in a reasonably readable manner. It then waits for a key press before going back to the Edit menu.

Now, truthfully, there is no need for this routine in the final version of the program. The indexing bugs seem to be gone, data moves smoothly, and loops keep on looping. But we thought it might be educational for you to see how we approach the debug process: carefully and with a lot of extra displays.

Well, after we've created a BlackBook file and added several records, we may notice that the file is getting full. Time to expand the file and make room for more phone numbers, right? Right.

20000-20270 Actually, this increase file size routine is almost identical with the Create a BlackBook file routine. The major difference is that we use the information about file space left on the disk (and the user's response to our query) to append a chunk of file to our existing. The Makeindex routine, discussed above, does all the work. Now you may notice why Startrec and Maxrec and Rec were all set up before the call to Makeindex in ADD. By doing so, we need only use other appropriate values to properly call the same routine here in Increase.

The only other possibility provided for here is the case of the clobbered index file. There are four ways the index file could become invalid: (1) Power to the computer goes off before the file is Closed or the disk is some how damaged. (2) The program crashes with an error. (3) You erase some records you didn't mean to. (4) You COPY the data file to another disk so that the NOTE pointers are no longer valid.

No matter what the cause, the Fix/Recreate Index routine will cure all ills. In the case of deleted records, it gives you a chance to recover them (so long as you didn't ADD a name after doing the accidental ERASE).

25000-25110 Again, we show the user what BlackBook files are on the disk and allow him/her to choose one. We prepare the screen for·some messages and fill the index string with $FF characters.

25130, 25540 Don't you wish BASIC XL had a function which would detect the end of a file? Well, it doesn't, but the PEEK() which controls this loop functions as one just fine.

25140-25180 We simply figure out where we are at in the data base file, get the record from disk (line 25160 would have to change if you add more fields to each record), and create a valid key, consisting of the separator byte, the record name, and the NOTE info.

25190-25310 Remember how we zapped the last byte of the PHONE$ string when we erased a record? Here's where that pays off. If such a record is detected, FIX gives you a chance to "undelete" it.

25320-25410 If the user wants to undelete the record, we change that magic character in PHONE$ to a space. If not, we change all the fields (and the record's name in the index string) to filler bytes. In any case, we write out the modified record. Lines 25390 and 25400 are necessary to avoid a

false end-of-file indicator (produced because of a bug in DOS) when writing the last record.

25430-25510 This part's almost easy: If the record found is a filler (blank) record, we simply add its pointer info to the end of the index string. If the found record is a real·one, we have to put its name in the proper place in the index string. Look at that! A call to our old friend, InsertKey, just exactly as if we were adding a new record.

25520-25530 Since we have to count the number of records in the file anyway, why not give the user some thing to watch as we work.

25550-25600 Funny how this code resembles that at the end of the Edit Menu exit and the end of the Makeindex routine. Maybe we need another subroutine just to write out the completed index file.

There will be a quiz tomorrow.

Whew! Did you get through all that? If so, then you are ready to convert BlackBook to your own needs.

Several fairly simple improvements would increase the usability and safety of the program dramatically. We leave them as exercises for you:

1. There's not a single TRAP in this entire hodgepodge. May we suggest TRAPping at least the most dangerous sections, such as where we create file, etc.

2. The Edit Menu is missing one obvious and important choice: Change (edit) an existing record. No good reason for the omission other than the fact that it seemed unnecessary in a demo program.

3. Cut the program up into pieces, chaining between them via RUN, so that the index string can be bigger.

4. Use a larger key. Change the file to a mailing list file (add field info in all the places we noted) and use the zip code plus first two letters of last name as the record name for the index string.

5. Use this basic program for something we didn't think of. Tell us about your efforts.

9.2.1Ø MAKEAUTO

We have received many requests for this program. Its purpose is quite simple: it creates an AUTORLN.SYS file for use with BASIC XL. More importantly, it allows you to specify one or more commands or statements which BASIC XL will execute on power-up.

We will not explain this program on a line-by-line basis, because the bulk of the program is so simple. It simply allows you to type in one line after another until you either enter a blank line (RETURN only) or you run out of room (you are allowed up to 159 characters, including RETURNs). It then writes out a new AUTORUN.SYS file by (1) reading the machine language program, including the run address, from some hex data statements and then (2) writing out your commands in a format acceptable to DOS's binary file loader.

Perhaps the only other thing worth mentioning is the fact that your commands are written out backwards (the FOR loop of lines 77Ø to 79Ø) to make the job of the machine language program easier. When AUTORUN.SYS is loaded by DOS, your backward commands will start at location $Ø6Ø1, preceded by a byte containing their total length less one (line 75Ø). Again, this is all to make the machine language program smaller and simpler.

Normally, we use AUTORUN.SYS to just cause BASIC XL to RUN our menu program. In other words, we respond to this program's prompt with

```
RUN "D:MENU.BXL"
```

However, you may choose any commands you wish. For example, suppose you had a very large program you wished to run on power up, but you want the user to know that the loading delay was normal. There are two solutions to that: (1) Have AUTORUN.SYS run a small program which simply prints a "please wait" message and then chains to the larger program, (2) Let AUTORUN.SYS do all the work, by answering its prompts like this:

```
GRAPHICS 16:POSITION 4,11
PRINT#6;"please wait"
RUN "D:MYPROG.BXL"
```

Why not? About the only statements you can't use via AUTORUN.SYS are those which might affect page six (e.g., POKEs) or the device handler table (at $Ø31A). Try it out yourself.

9.3 BASIC XL Extended Statements

9.3.1 How to Install the Extended Statements

Because BASIC is usually an interpreted language, it is no more flexible than the keywords with which it is endowed. When we at OSS designed BASIC XL, we wanted a true interpretive BASIC with a reasonable amount of power and speed. However, we also wanted a degree of flexibility unmatched in most versions of the language. Hence the ability to add statements to the language was included, even though no such "extended" statements existed. Until now!

This release of The BASIC XL ToolKit includes six new extended statements for you to use in your own programs. The statements added fall into two groups: (1) procedure calls and (2) string array sorting. Before describing the new statements (in sections 9.3.3 and 9.3.4, respectively), we need to discuss how these extended statements are added to BASIC XL.

If you request a directory of the reverse ("flip") side of your BASIC XL ToolKit disk (via BASIC XL's DIR command), you will find the file

```
EXTEND.COM
```

and it is this file which contains the code which implements the extended statements.

There are several ways to begin using the extended statements. The easiest way is to simply duplicate that flip side of your ToolKit disk and boot the resultant copy. (Again, please don't use your original disk for anything other than making duplicates. Thank you.)

The reason booting that flip side works is that, in addition to EXTEND.COM, we have provided you with an AUTORUN.SYS program which incorporates both the extensions (identical code to that in EXTEND.COM) and a BASIC XL command invoker identical to that provided by MAKEAUTO.BXL (see section 9.2.10). In the version on your disk, we have given this MAKEAUTO equivalent only one command:

```
RUN D:EXTEND.BXE
```

In turn, EXTEND.BXE is a very, very short program. We list it here in its entirety:

```
10 Graphics 18 : Position 2,12
20 Print #6; "...please wait..."
```

```
30 Move $570,$C4,4
40 Run "D:MENU.BXL"
```

The only important line here is line 30, the MOVE statement. NOTE CAREFULLY: even after the extended statements have been loaded into memory, they must be made available to BASIC XL. This is accomplished by placing pointers to their execution and syntax tables in $C4-$C5 and $C6-$C7. This has to be done after BASIC XL issues the Ready prompt, because BASIC XL always clears these locations to zero upon a coldstart (e.g., at power-on). Note the other implication of this: if, later, you convince BASIC XL to undergo a coldstart (either by exiting to DOS and performing a LOAD of some kind or, as some programs do, by POKEing the warmstart flag off), you must once again perform this MOVE or the extended statements will not be available to you. (Actually, if you exit to DOS and LOAD or run some program, the chances are good that you should then LOAD EXTEND.COM again, since most disk-based programs will overwrite the memory used by the extensions.)

Another way to implement the extensions was just hinted at: you may, from virtually any DOS, simply LOAD EXTEND.COM and then enter the BASIC XL cartridge. If you are using a menu-driven DOS, choose the appropriate menu options to do the LOAD and enter the cartridge. If you are working with OS/A+ or DOS XL, you may simply type

```
EXTEND
CARTRIDGE
```

in response to the D1: prompts (and, in turn, these commands could be part of a STARTUP.EXC file -- see your DOS XL manual). If you enter BASIC XL in either of these ways, you will be presented with the Ready prompt. In order to use the extended statements, you will have to use a MOVE $570,$C4,4 command as was given above.

The final way to implement the extensions which we will explore here is a variation on the first one. Simply replace the program EXTEND.BXE with your own program of the same name. If you keep the MOVE statement in your program, and if it is executed before you use any extended statements, this will work just great. Probably the easiest way to customize EXTEND.BXE to your own purposes would be to simply change the name of the program to RUN in line 40.

Remember: the DOS given you on this disk has neither menu nor command processor. It is only capable of booting a disk with an AUTORUN.SYS file present. You may, however, COPY all or some of the files on this

disk to another one which has your preferred version of DOS already on it.

Without further ado, then, let us proceed toward the descriptions of the extended statements.

9.3.2 Abbreviations Used In Formal Statement Definitions

The following are the abbreviations used in the formal format definitions of the following sections (an abbreviation marked with an asterisk is new; others are consistent with the BASIC XL Reference Manual):

avar — **a**rithmetic **var**iable, neither a string nor an array.

Examples: TOTAL   I   J   XØ

svar — **s**tring **var**iable, either a string array or simple string, distinguished from an avar by a trailing dollar sign.

Examples: NAMES$ SA$

Note that one, two, or three subscripts are often used between the parentheses following an svar. For the special case of an svar used to satisfy the requirement for a pvar or cvar (see be low), no paren theses may be used.

savar — **s**tring **a**rray **var**iable, same format, etc., as svar but must be a properly dimensioned array.

mvar — **m**atrix **var**iable, numeric array, distinguished from an avar by a trailing left parenthesis.

Examples: VALUES()   SCORES()

Note that one or two subscripts normally appear between the parentheses following an mvar. For the special case of an mvar used to satisfy the requirement for a pvar or cvar (see below), nothing may appear between the parentheses.

aexp — **a**rithmetic **exp**ression, any valid combination of numeric value, operators, etc.

Examples: 33   7+VALUE   SCORE(3*J)

* rparm **r**eceiving **par**a**m**eter, either an avar or an exclamation point followed by an svar or mvar.

  Examples: TOTAL   !NAMES$!VALUES()

* cparm **c**alling **par**a**m**eter, either an aexp or an exclamation point followed by an svar or mvar.

  Examples: 29*SIN(3Ø)   !TEMP$!AMAX()

slit **s**tring **lit**eral, a string of characters enclosed in quotation marks.

  Examples: "TOTALIZE"   "Test-->>"

* pname **p**rocedure **name**, used to identify a procedure, always consists of only an slit.

* cname **c**alling **name**, used to name a procedure to be CALLed, may be either a slit or svar. If a svar is used, it may not be a string array and may not use any subscripts.

Remember: words in a format definition which are given in all capital letters (e.g., USING) must be entered exactly as shown. Items in square brackets are optional. Items with ellipses following may be repeated as desired. Example: rparm [,rparm,...] implies that you may use one or more receiving parameters.

9.3.3 Procedure Blocks and Related Statements

Before describing the individual statements, we present an overview of PROCEDUREs in BASIC XL.

If you have programmed at all in any dialect of BASIC, you have used the GOSUB statement and its companion, RETURN. For example, you might see a program which looks something like that which follows. (This program is for demonstration purposes only, but it is a fairly amusing little thing to spring on an unsuspecting friend.)

```
2Ø Value=1ØØ
3Ø Min=1Ø : Max=9Ø : Gosub 1ØØ
4Ø Result1=Num
5Ø Min=1Ø*Value : Max=9Ø*Value : Gosub 1ØØ
6Ø Result2=Num
7Ø If Result2 > Value*Result1 Then 9Ø
8Ø Print "You appear to be conservative in nature." : End
9Ø Print "You seem ready to take risks." : End
1ØØ Rem THE SUBROUTINE
```

```
110 Print: Print "Please give me a number between "; Min
120 Print" and "j Max ;
130 Input ", inclusive > ",Num
140 If Num>=Min And Num<=Max Then Return
150 Print "Can't you read? That number is"
160 Print" out of the range I gave you."
170 Goto 100
```

And, in a small program like this one, that usage of GOSUB may be just fine. As programs get larger, though, lines such as GOSUB 3250 become less and less meaningful. Atari BASIC (and thus BASIC XL) allows you to do something like this:

```
10 Let Getinrange=100
20 Value=100
30 Min=10 : Max=90 : Gosub Getinrange
   (etc.>
```

Do you see what we did? By giving a name to the subroutine, we can make our code more readable. A disadvantage to this method is that BASIC XL (in common with Atari BASIC) allows only 128 unique variable names. Using a variable like this to name a subroutine diminishes the pool of available names. This, then, is the first advantage of BASIC XL's new procedures: because we use a literal (quoted) string to name them, we need waste no variables! For example:

```
20 Temp=100
30 Call "Get In Range" Using 10,90 To Result:
50 Call "Get In Range" Using 10*Temp,90*Temp To Result2
70   If Result2 < Temp*Result1 : Type$="conservative"
80   Else: Type$="a risk taker"
90   Endif
95 Print "You Seem to be "; Type$; " by nature." End
100 Procedure "Get In Range" Using Min,Max
110 Local Temp: Temp=1E90
120   While Temp<Min Or Temp>Max
130     If Temp<>1E90 : Print
140     Print "Can't you read? That number is"
150     Print "   out of the range I gave you."
160     Endif
170   Print : Print "Please give me a number between "; Min
180   Print "   and "; Max ;
190   Input " , inclusive > ",Temp
200   Endwhile
210 Exit Temp
```

Confused? Not too surprising. Let's take a look at the new lines a step at a time. First, in line 30, note the CALL to the PROCEDURE named "Get In Range" (which starts at line 100). Note how clear that CALL is, since

we can use any characters we like in the string. That's pretty easy, right?

But what about that USING which appears in both the CALL and PROCEDURE statements? In line 3Ø, we are "Using" values of 1Ø and 9Ø. But in line 1ØØ, we are "Using" the variables Min and Max. Isn't that neat? We didn't have to do the assignments to the variables before we called the subroutine: CALL does the work for us! It automatically moves the values (1Ø and 9Ø) into the corresponding variables (Min and Max). This is called "passing parameters" to a PROCEDURE.

It gets better. Notice the EXIT statement of line 21Ø. It specifies a value (the contents of Temp) which is to be placed in to the variable Result1 that follows the TO in the CALL statement. That's reasonable, right? If you Can "pass" parameter values, you should be able to "return" parameter values.

But doesn't using the variable Temp in the procedure subroutine wreak havoc on its later use in the main program (e.g., in line 6Ø)? Ah, but there's line 11Ø, with ts deceptively simple-looking LOCAL statement. Between the use of LOCAL Temp and the EXIT statement, the old value of Temp is saved for you. When EXIT is executed, all LOCAL variables are automatically restored to their previous values. Wow! And Whew!
The example we just worked through used all of the new PROCEDURE-oriented extended statements:

PROCEDURE
CALL
LOCAL
EXIT

By no means, though, did we use all of the capabilities of these 5tatements. In addition to the formal definitions which will follow, we will present further examples both in the text and in programs on the disk.

We have presented these statements before the formal definitions because they are all closely related, and we felt that having a small but effective demonstration of their use would make it easier to understand the definitions.

9.3.3.1 PROCEDURE (PROC.)

Format: PROCEDURE pname [USING rparm [.rparm...]]

Examples:

```
1ØØØ Procedure "Calculate Pay" Using Hours,Rate,Taxtable()
 387 Procedure "Print Msg" Using !Msg$
4Ø4Ø Procedure "Quit"
```

The PROCEDURE statement is the nucleus around which the other statements in its group are built. It is used to define the beginning of a subroutine which is intended to be executed via a CALL statement.

A PROCEDURE must be given a name, which may be any set of ATASCII characters enclosed in quotation marks, the number of characters being subject only to the limitation that the entire line must be of legal length. Note in the examples above how spaces have been used in the PROCEDURE names to add clarity to the program. As a matter of good programming style, you should make the names as self-explanatory as possible, shortening them only if you begin to run out of memory.

When a CALL statement is executed, it places an entry on the Run-Time Stack (the same stack used by GOSUB, FOR, WHILE, and their partners). This entry serves to identify the fact that a PROCEDURE statement has been encountered, and its subroutine (which we will here call the "procedure block") is now in control. When the PROCEDURE statement itself is executed, then, it ignores its own name and does nothing further to the Run-Time Stack. Unless, that is, the user has specified that one or more parameters are being passed via the USING keyword.
If USING is coded, it must be followed by one or more variable names. If the variable names refer to string variables, string arrays, or numeric arrays, the name must be preceded by an exclamation point (!). No matter which kind(s) of variable(s) is/are used, when PROCEDURE is executed, their current "values" are pushed onto the Run-Time Stack. Then, after the values have been pushed, the new values as specified in the CALL which invoked this procedure block, are copied into these same variables.

When working with simple numeric variables, this is a fairly straight-forward process. Take the following set of statements as an example:

```
1Ø Junk=2Ø
2Ø CALL "Test" USING 12*17
3Ø Print Junk
4Ø End
```

```
   ...
70 PROCEDURE "Test" Using Junk
80 Print Junk+Junk
90 Exit
```

In this example, when the PROCEDURE named "Test"·at line 70 is invoked and the statement is executed, the current value of the variable Junk (20, as assigned in line 10) is pushed on the Run-Time Stack. Then the value of the expression (12*17, or 204) is copied into Junk. Any subsequent references to Junk will find that it contains this new value. For example, the Print of line 80 will display the value 408.

The effect of pushing the prior value of Junk is simple: when the EXIT statement (line 90) is executed, it will discover the value that was pushed on the stack and restore Junk to its prior condition. Thus the Print of line 30 will display the value 20. (The EXIT statement is discussed in more detail in section 9.3.3.4.)

The purpose of all this pushing may be less clear. First, by "reusing" the variable name Junk in our procedure block, we are conserving our precious names (remember, we are allowed only 128 different names in a program). Since the value of the variable is restored on EXIT from the block, we need not worry about changing it within the block. Second, and perhaps more difficult to grasp from this simplistic example, we are able to pass values "into" the procedure block without having to be aware of what names are used within it. The example which introduced this chapter shows this feature to some advantage and also serves to demonstrate how the resultant code can be both smaller and more readable.

For strings and arrays used as PROCEDURE parameters, the methodology is the same, but the results are more complex. The difficulty lies in understanding just what is the "value" of a string or array. In Atari BASIC and BASIC XL, the value of any variable is the content of its entry in the Variable Value Table. This table reserves eight (8) bytes per variable and consists of a flag byte, the variable's number (0 through 127), and six bytes of "information".

In the case of simple numeric variables, the information is the numeric value of the variable, expressed in an internal floating point form. (You may consult the Atari Technical Manuals or COMPUTE!'s Atari BASIC Source Book for much more detail on the structure of these and other tables.)

For string and array variables, the flag byte indicates that the "information" describes the location and

characteristics of the contents of the variable. For example, a simple string variable needs information about its address (within string/array space), its dimension, and its current length. The string itself (the "contents" of the variable from an external point of view) is located at the given address. Arrays (both string and numeric) need an address and two dimensions instead; but, again, the actual "contents" are found at the given address.

Thus, when we push the "value" of a string or array variable on the Run-Time Stack, we are pushing this information about where the actual contents are located in memory. Similarly, when we copy a value passed by the CALL statement into one of these variables, we are not copying the actual string or array. Instead, we are copying the address, dimension, etc., as appropriate. Consider this sequence:

```
10 Fun$="Swimming is fun." : X$ = "Right?"
20 CALL "What Fun" USING !Fun$
30 Print Fun$, X$
40 End
   ...
60 PROCEDURE "What Fun"·USING !X$
70 Print Fun$, X$
80 X$(1,5)="Laugh"
90 EXIT
```

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

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

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

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

Do you see why? If the value of Fun$ is copied to X$, then the address of the contents of Fun$ is now in X$'s address entry with its value in the variable table. Thus, any change we make in the string pointed to by X$ affects the memory at that address and thus affects the contents of Fun$. Complicated, yes?

A similar action takes place when a string array or numeric array is passed as a parameter: changes in the contents of the PROCEDURE's parameter affect the contents of the CALLer's parameter.

Technical Note: In computer lingo, simple numeric variables are passed to a procedure block via a "call by value". Arrays and strings, on the other hand, are passed via a "call by reference". The exclamation point required by the syntax of the extended statements can be used as a reminder that these are calls by reference, something not hitherto seen in BASIC XL. (Actually, the exclamation point is necessary so that the expression evaluator can make the distinction between an expression — which could, for example, start with a string or array reference — and one of these special calls by reference.)

9.3.3.1.1 Secondary Considerations
----

(1) You may, if you wish, pass too many numeric parameters to a PROCEDURE. BASIC XL makes no check for matching number of parameters. It does, however, insist on a type match. Thus this sequence will cause a "USING Type Mismatch" error:

```
     4Ø1Ø CALL "Gorp" USING 33
          ...
     72B9 PROCEDURE "Gorp" Using !A$
```

If the CALL passes too many parameters, the excess are ignored. If it passes too few, a numeric value of zero (Ø.Ø) is assigned to all remaining PROCEDURE parameters. This, in turn, can cause a type mismatch, since only numeric variables may receive a numeric value.

Exception to the last paragraph: If the CALL passes no parameters, BASIC XL does nothing at all to the parameter passing area. This is on purpose, since passing parameters takes time . Thus, even a PROCEDURE expecting only numeric parameter(s) may report a mismatch error, since it attempts to obtain those parameters from the miscellaneous data left in the parameter area. Generally, we recommend passing the correct number of parameters unless you have a specific purpose which can use the "default" feature to a real advantage.

(2) You must be careful when changing the value of a simple string passed as a parameter. Recall that the length of a CALLing string variable is found in its variable value table entry, and that the entry is copied intact to the PROCEDURE's string variable. If you then change the length of the string within the procedure block, it will indeed change the PROCEDURE variable's entry. However, when you EXIT, the entry is not automatically copied back to the CALLer's variable! This can produce some bizarre results.

To demonstrate: modify line 8Ø of the last example program to read

```
80 X$="Laugh" : Print X$
```

Not surprisingly, the new Print in line 8Ø shows us that the contents of X$ are simply "Laugh". However, look at the display resulting from line 3Ø:

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

Do you see the problem we warned of? Changing X$ in line 8Ø changed the memory at the address which Fun$ also used for its contents, but it did not change the length of Fun$. Presumably, this could be a feature under the right circumstances, but there are stranger consequences possible. For example, try changing line 8Ø to read

```
80 X$="XXX"
```

Now line 3Ø's Print will display

```
XXXmming is fun.   Right?
```

which is almost surely not we wanted.

One solution to this situation is simply to avoid changing a passed string within a procedure block. This may not be satisfactory, though, so we have provided another mechanism which you can use to circumvent the problem: Change lines 2Ø and 9Ø in the original program to read

```
20 CALL "What Fun" USING !Fun$ TO !Fun$
90 EXIT !X$
```

EXIT will be discussed in more detail in section 9.3.3.4, but suffice to say that this sequence guarantees that the complete new value of X$ is copied back to Fun$. On this same topic, you may be relieved to know that the difficulty with length does not exist with arrays, either of strings or numeric values.

(3) One way to get in real trouble with either strings or arrays is to pass back (via EXIT) one which was not passed in as a CALLing parameter. Examine the following program excerpt:

```
100 CALL "Oops" To !A$
110 CALL "Oops" To !B$
120 Print AS,B$ : End
    …
300 PROCEDURE "Oops"
310 Input "Type something: ",Line$
320 EXIT !Line$
```

If you enter and RUN this program, giving a different response each time you are prompted, you will be surprised at the results of the PRINT of line 12Ø: A$ and B$ will be identical (up to the length of the shorter), taking on the value of your second INPUT. If you recall our discussion of what actually gets passed when a string or array is involved, this seemingly bizarre result can be explained.

When you pass LINE$ back to the CALLer, you are actually transferring the contents of LINE$'s variable value table entry to first A$ and then to B$. But that table entry consists (among other things) of LINES's address. Thus you end up with all three variables pointing to the same piece of memory!

Once again, the proper solution is to pass a string both in via USING and back out via EXIT. For arrays (of either strings or numbers), you need only pass the value in, since anything the PROCEDURE does to a parameter array is properly reflected in the CALLer's original value(s).

The only way you can get in trouble with arrays is if you pass an undimensioned array to a procedure block which then dimensions it. Unless you pass back the "value" via EXIT (similar to the fix for strings just given above), the space dimensioned within the block is simply lost, since no variable will any longer be referring to it via the address portion of its entry in the variable value table.

When in doubt, then, pass strings and arrays both ways. It can't hurt. It may help.

(4) Finally, another caution. A PROCEDURE must be the first statement on a line. CALL can not find a PROCEDURE if is not at the beginning of a line. Strange and wondrous and woefully unpredictable things can happen if you violate this rule.

Similarly, you should never allow a program to "fall through" to a PROCEDURE. Always make sure that the program immediately preceding each PROCEDURE finishes with a GOTO, STOP, END, RETURN, or EXIT statement. We recommend grouping all procedure blocks at one spot in your program and ensuring that they are preceded by an END statement.

9.3.3.2 CALL

```
Format:   CALL cname [USING cvar[,cuar...]] [TO pvar[,pvar...]]

Examples:  1Ø CALL "Test"
          72Ø CALL "Totals" USING !Values() TO Sum
          8ØØ CALL "Get Num' TO Number
          1ØØ CALL Proc$ USING 7,!A$ TO Result
```

The CALL statement has been discussed and demonstrated in both the introduction to this chapter and in the explanation of the PROCEDURE statement (section 9.3.3.1). In this section, then, we will not dwell on such things as the mechanics of parameter passing. Rather we will discuss the subtleties of the CALL statement itself.

First, unlike a PROCEDURE statement, the name specified by a CALL may be contained within a string variable instead of being a string literal (see the last of the above example lines). However, you have no other choice of format than that shown. You may use neither a substring nor an element of a string array as a CALLed name. (This stricture was necessary for consistency, in order to allow the syntax to be as close as possible to that of PROCEDURE. The alternative was using a comma instead of the word USING.) This is not an onerous restriction, though, as the great bulk of all calls will probably be made with litera1 strings.

For those rare occasions where you wish to choose one of several PROCEDUREs based on the value of some index, may we suggest a program format similar to the following:

```
3Ø Input "Give me an index > ",Index
4Ø Name$=Proc$(Index;) : CALL Name$
```

Remember, also, that the name which you CALL with (whether literal or variable) must match exactly that given in a PROCEDURE statement. All characters are considered in the match (including leading or trailing spaces), with upper case, lower case, and inverse video all distinct.

Second, we remind you of the possible problem associated with using a string variable as a CALLing parameter (if its length is modified in the procedure block, the length change is not visible to the CALLer -- see section 9.3.3.2). Generally, it is good form to always code a simple string variable as both a calling and returning parameter, thus:

```
999 CALL "Invert String" USING !Gorp$ TO !Gorp$
```

Similarly, any array which may not be dimensioned at the time of the CALL should receive the same treatment. Recall our earlier cautions, also: DIMensioned arrays need not be passed back to the CALLing routine, but they must be passed in as parameters.

9.3.3.2.1 Secondary Considerations

The number of levels you may nest CALLs is limited only by the amount of FREe memory left in your system which may be used by the Run-Time Stack. Like GOSUBs and WHILEs, each CALL uses four (4) bytes of Run-Time Stack space. Each parameter passed (either expression value or string/array reference) occupies 12 bytes. A demonstration of the implications of these facts may be found in the example programs in section 9.4 (see especially the FACTORIAL program).

CALLs are slow when compared to GOSUB line-number in BASIC XL's FAST mode. However, when compared to normal GOSUBs in slow mode, they may actually be just a bit faster if they do not pass parameters. Parameter passing can, indeed, slow things down remarkably. But, when you compare it to the method of doing several assignments before a GOSUB followed by one or more afterward, it may actually save time in some situations.

Within a CALLed procedure block, you must never attempt to POP the parameter variables. You can cause a sytem crash if you POP a variable with the wrong value. Only if a procedure block has neither parameters nor LOCAL variables may you safely POP the CALL itself. We recommend that you do not use POP anywhere in a procedure block unless absolutely necessary.

9.3.3.3 LOCAL

```
     Format:   LOCAL avar [,avar...]

     Examples: 73Ø LOCAL Temp1
               137Ø LOCAL Sum,N,Count,Misc
```

The LOCAL statements has been provided to allow you more flexibility in your programming. While the parameter received by a PROCEDURE are automatically made local to that procedure block, there are many times when you need a simple variable to hold a temporary value, such as the result of a calculation, a flag, etc. LOCAL gives you such temporary variables.

LOCAL works in a very simple fashion. When a LOCAL statement is excuted, all simple arithmetic variable names (no strings or arrays allowed) following it are "pushed" onto BASIC XL's run-time stack (the same stack which receives GOSUBs, FORs, CALLs, etc.). Then, when a subsequent EXIT is encountered, all such LOCAL variables are pulled back off the stack and put in their original places. The effect of this is simple yet powerful: within the bounds of LOCAL and EXIT, you may change the value of any of these variables to your heart's content without worrying about whether some other routine in your program is using a variable with the same name.

A simple example will help:

```
     1Ø Test=1234567 : Print 1Ø,Test
     2Ø Gosub 4Ø : Print 2Ø,Test
     3Ø End
     4Ø Local Test : Print 4Ø,Test
     5Ø Test=Ø.54321 : Print 5Ø,Test
     6Ø Exit
```

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

```
     1Ø          1234567
     4Ø          1234567
     5Ø          Ø.54321
     2Ø          1234567
```

Let's examine that program a little closer. First, line 1Ø is simple enough. We just assign a value to the variable and verify that it has been accepted. In line 2Ø, we first GOSUB to a routine and then again display the contents of our variable. Note that in the program's running this PRINT of Test is the last thing executed (other than END).

Line 4Ø, then, begins the interesting part of this program. We declare that Test is a LOCAL variables and, once again, display its value. Line 5Ø is a repeat of line 1Ø except that we assign a different value to our variable. Note that the PRINT verifies our change. Finally, in line 6Ø, we use another new statement, EXIT, to restore our variable to its original value, as shown by the PRINT in line 2Ø.

Once again, the point of all this was that our subroutine (lines 4Ø through 6Ø) could do what it liked with the now-LOCAL variable without affecting its value in the rest of the program.

9.3.3.3.1 Secondary Considerations

Some things are made obvious in the above program which
(1) LOCAL does not have to be used in conjunction with
(2) The value of a variable which is made LOCAL does
because of the push onto the Run-Time stack. We will
points in order.

The fact that LOCAL may be used with GOSUB-typ" subroutines is not an accident. EXIT was specially constructed to examine what invoked its subroutine and handle the returning condition appropriately (either GOSUB or CALL only, though). This small fact alone may allow you to change many programs to use LOCAL without the need to modify all GOSUBs·to CALLs.

Also, there are occasions where it could be advantageous to use GOSUB instead of CALL. In particular, GOSUB to an absolute line number is significantly quicker when your program is·in FAST mode than any other type of subroutine access. (A mild warning, though: LOCAL does occupy precious processing time, so you may do best to use truly unique variable names in a routine which must be super fast.)

Our second point, the fact that variables do not change value when they are made LOCAL can actually be used to advantage in a few cases. Try the following small example program:

```
1Ø Input "An integer greater than 1, please >> ",N
2Ø Sum=Ø : Gosub 5Ø
3Ø Print "The sum of integers from 1 to ";N;" is ";Sum
4Ø End
5Ø Local N
6Ø Sum = Sum+N
7Ø If N=1 Then Exit
8Ø N=N-1 : Gosub 5Ø
9Ø Exit
```

To follow what happens here, assume that we choose a value of 3 for our integer. The first time lines 5Ø through 7Ø are executed, then, Sum will take on the value of 3 and, since N is not 1, we continue on to line 8Ø. There N is given a value of 2 (one less than its current value), and we again call the subroutine at line 5Ø.

The second time through, the same things happen: Sum acquires a value of 5 and we do not yet do the Exit of line 7Ø. In line 8Ø, N's value changes to 1 and line 5Ø is called once again.

This third time performing the same lines sees lines 5Ø and 6Ø performing as before, with Sum getting a new value of 6. In line 7Ø, though, since N now has a value of 1 we do take the Exit. We return to the Gosub of line 8Ø, fall through to line 9Ø, return to line 8Ø again, fall through to line 9Ø again, and (at last!) return to the original Gosub of line 2Ø.

9.3.3.4 EXIT

```
     Format:   EXIT [ cparm [,cparm...] ]

     Examples: 39Ø EXIT 1Ø*Maxvalue
               799 EXIT Flag,!Names$
               2499Ø EXIT !Inverse() ,Rows,Columns
               835 EXIT
```

If you have been reading this instruction manual in front to back order, you have encountered several examples of the use of EXIT by now. If you have not, we refer you to sections 9.3.3, 9.3.3.2, and 9.3.3.3 for some illustrative examples.

Just as Return is a partner to Gosub, so is Exit a partner to Call. Every Procedure which you invoke via Call must end with an Exit statement.

Exit performs three functions, in the following order : (1) If there are any parameters after the Exit keyword, they are placed into BASIC XL's parameter-passing area, for use by the TO-keyword's processing (which is, in turn, part of the work which Call does). (2) If there are any variables on the run-time stack (either as a result of using a local statement or needing to save the parameter variables of a Procedure), Exit must restore them to their proper places in the variable value table. (3) Exit checks to see whether the current subroutine was invoked via Call or Gosub. If via the latter, Exit simulates the action of a Return statement; otherwise, it performs the special processing needed to allow TO to access its parameters (if any).

9.3.3.4.1 Secondary Considerations

In common with the other stack pulling statements (Return, Endwhile, Next), if Exit discovers a For on the Run-Time stack which doesn't "belong" there, it ignores it (e.g., it "throws it away") and tries the next entry on the stack. For example, the following program will not cause an error:

```
10 Gosub 50
20 End
50 Rem === Subroutine ===
60 For I=1 To 5
70 Exit
```

Even though the For loop started in line 60 has not finished (and is thus still sitting on the stack), Exit has no trouble finding that its subroutine was called via the Gosub of line 10.

On the other hand, this program will cause a 'nesting' error because While can only be terminated by Endwhile!

```
10 Gosub 50
20 End
50 Rem === Subroutine ===
60 While 1 : Rem (a never ending loop)
70 Exit
```

Another thing to be careful of is that no error will result if an Exit statement tries to pass parameter values back to a Gosub. Instead, they are simply ignored. (The reason for this, again, is that the cartridge BASIC XL is not prepared for such things, so it does not check for them.)

Similarly, if you pass back too many parameters to a Call, the excess ones will be ignored. This design allows a single Procedure to serve more than one function, returning more values to some Callers than to others. Remember, though, that all parameters expected by the TO portion of a Call statement must be matched by type by the parameters of Exit (e.g., a string variable to a string variable, a numeric expression to a numeric variable). The matching needed is the same as that needed by parameters passed to a Procedure via a Call. See section 9.3.3.1 for more details.

Since you can never properly Pop variables, you may not use Pop in a subroutine which uses either local variables or Procedure parameter variables. Thanks to the fact that Exit may return a parameter value, we find little need to use Pop in these circumstances anyway. A better method is illustrated here:

```
 10 While
 15 Call "Demo 1"
 20 Endwhile
    ...
 50 Procedure "Demo 1"
 55 N=Random(8) : Call "Demo 2" Using N To Flag,Inverse
 60 If Flag Then Exit
 65 Print "The inverse of ";N;" is ";Inverse
 70 Exit
    ...
 85 Procedure "Demo 2" Using Value
 90 Trap 95 : Exit 0,1/Value
 95 Exit 1
```

The trick in this program is embodied in lines 90 to 95. In line 90, we first set up a Trap to line 95, in case an error occurs. But where can an error occur? Certainly not in the evaluation of the zero following the Exit. But what about when we evaluate 1/Value? If Value is zero, this expression will cause overflow, an error condition. If the error occurs, the Trap will send us off to line 95, where we simply return the flag value of one, indicating failure.

Line 60 is where we check the value of the returned flag. If it is non-zero, we immediately Exit rather than displaying the results. Do you see why this is cleaner than using a Pop statement? Aside from the fact that the flow of the program becomes much more readable, we could add man y local variables at any point in this program without adversely affecting its functioning.

This concludes our presentation of the BASIC XL ToolKit extended statements which relate to Procedure blocks. See also section 9.4 for discussions of the example programs provided on your ToolKit disk.

9.3.4 Sorting String Arrays

Apart from the PROCEDURE blocks described in section 3.3, the only extended BASIC XL statements included with this ToolKit are those which allow you to easily sort a string array. There are two such statements, SORTUP and SORTDOWN, which are described formally in sections 9.3.4.1 and 9.3.4.2 (respectively). However, since both sorting statements have many foibles in common, we thought it best to begin with some comments and hints about their use.

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

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

This program simply allows you to INPUT five strings, sorts them, and then shows the sorted order. At this time, we would like to suggest that you boot a copy of side 2 of your master ToolKit diskette. Then type in this program and try it out. (Keep it around. We will use it more later.) Give several sets of common and uncommon words as answers. Note how neatly it sorts the words into ascending order.

Or does it? Try entering some words in upper case and some in lower case. What happens? Does it surprise you to find that "ZOO" comes "apple"? Actually, the reason for this behavior is readily understood once you realize that SORTUP works on characters using ATASCII ordering (ATari version of ASCII, the American Standards Code for Information Interchange -- how's that for a mouthful). For a list of ATASCII codes as they relate to your computer's keyboard, see Appendix D of the BASIC XL Reference Manual.

Even if we restrict ourselves to the "printable" characters in the ATASCII set (usually the numbers, upper and lower case letters, and standard typewriter-style symbols -- codes numbered 32 through 124 in the manual), we find no real help. Numbers come before upper case letters which come before lower case letters, but symbols are intermixed in no real useful fashion.

Because the effects of this hodgepodge ordering may not be desirable in a sorted list, you may wish to limit a SORTUP or SORTDOWN to work with only part of each element of a string array. For example, if you have an array where each string within it contains both a person's name and their phone number, you may wish to perform a sort based solely on names. Further, to ensure that the sorted order is consistent, you may wish to ensure that the names being sorted are stored as upper case letters only.

Fortunately, the design of SORTUP and SORTDOWN is good enough that sorting based on "fields" (portions of each element in the string array) is extremely easy. And, while BASIC XL does not provide a built-in method of obtaining upper-case-and-non-inverse-video-only strings, it isn't very hard to build a routine which will do the real work for you. For example, the following PROCEDURE converts all characters in its parameter string (not a string array) to non-inverse video and converts lower case letters to upper case:

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

For now, don't enter that subroutine.

Instead, let's investigate the concept of " fields", as mentioned above. Just change line 30 in that little program we typed in earlier so that a LIST gives you the following:

```
10 Dim Array$(5,20)
20 For I=1 To 5 : Input Array$(I;) : Next I
30 SORTUP Array$ USING j 3,5
40 For I=1 To 5 : Print Array$(I;) : Next I
50 Run
```

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

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

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

Now, since the SORTUP of line 30 refers to the entire array, String$, there is no need for the following parentheses (and, indeed, they are not allowed). Instead, the keyword USING tells BASIC XL that we will be working with only part of the array and/or its elements. In particular, the semi-colon following USING again serves as a reminder that the numeric expressions following it refer to character positions within an element (or, more properly when using SORTUP or SORTDOWN, within all elements) of a string array.

By the way, as a simple variation on what we have done so far, you might change line 39 to read:

```
30 SORTDOWN Array$ USING ; 3,5
```

Again, try it out. Not too surprised by the results? Good. The only difference between SORTUP and SORTDOWN is where the "top" of the sort (the "largest" string) appears.

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

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

The first change you will notice is in lines 20 and 25. Instead of blindly continuing to ask for INPUT until 20 items have been entered, the program only goes back for another if the length of the current item is non-zero. That means that you may stop entering items at any time

by hitting the RETURN key alone in response to any INPUT prompt.

And look at SORTUP in line 3Ø. Can you guess what Using 1,I-1 is for? That's right, only the first I-1 elements of the array will be sorted! And if, for some reason, you wanted to never sort the first element of the array, you could have coded

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

(Why would you ever do that? Well, maybe you keep special information about a file in the first "record" of the file, thus having the actual data start at the second "record".) In fact, you are not limited as to which elements may be sorted other than having to follow two rules: (1) The maximum element number to be sorted must be greater than or equal to the minimum element number. (2) Each number must be within the bound of the array, as dimensioned.

Naturally, we have to give you the last of the possible variations on SORTUP (and, similarly, on SORTDOWN). We won't explain this. Just type it in and try it:

```
30 Sortup String$ Using 1,I-1 ; 2,4
```

Now for some hints.

We already noted that it is probably a good idea to restrict the contents of a normal alphabetic field to upper-case, non-inverse characters only. Suppose, though, that you really want to sort some numbers. What can you do? A program such as the following will not work:

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

Why not? Well, try some numbers in response to the INPUT prompts and see what happens. May we suggest values of 1, 11, 111, 2, and 22 for your test. When we tried those numbers, BASIC XL told us that the order was

```
1
11
111
2
22
```

If you think about the ATASCII values of those characters (and they are characters, since they are in a string) for a bit, you will realize that those are the proper results. The problem, then, is to make numbers appear in a string in a fashion such that the sort statements can handle them.

We could present a complete solution here, but we leave that for a program on the ToolKit disk (called SORTNUM.BXL. We will, however, consider at least the case of sorting positive integers, which may cover all the cases you will ever need.

```
10 Dim String$(5,10)
20 For I=1 To 5 : Input N : String$(I;)="0000000000"
25 String$(I;11-Len(Str$(N))) = Str$(N) : Next I
30 Sortup String$
40 For I=1 To 5: Print String$(I;) : Next I
50 Run
```

We have altered line 20 and added line 25. The trick here is not too terribly obscure: We first fill the pertinent element of the string array with place-holding zeroes. Then we position our integer at the proper location within that field of zeroes. Since all numbers (as represented in ATASCII) are now the same length, it is only the significant digits which affect the sort process. Try it and see.
Note that there is no protection in this program to keep you from entering a number which is not a positive integer. Purists might add line 22:

```
22 If N<>Int(N) Or N<0 Or N>=1E10 Then Print "Bad number":Stop
```

And, if you prefer a neater looking numeric print-out, you can change line 40 to:

```
40 For I=1 To 5 : Print Val(String$(I;)) : Next I
```

We at OSS can see many uses for SORTUP and SORTDOWN. Again, we invite you to peruse the sorting demo programs on the ToolKit disk. Perhaps you can find a use for some of the techniques in your own programs.

9.3.4.1 SORTUP

Format: SORTUP savar [USING [aexp TO aexp] [;aexp,aexp]]

```
Examples: SORTUP Stringarray$
          SORTUP Array$ USING Min TO Max
          SORTUP X$ ; 1,4
          SORTUP X$ Using 5 To 10 ; 4,8
```

This statement will sort selected elements of a specified string array in ascending order, based on the contents of a selected portion (a "field") of each element of the array. Unless otherwise specified by the user, the field of each element which forms the basis for the sort shall consist of the entirety of each element. Unless otherwise specified by the user, all elements of the array will be selected to be sorted.

The user may choose the beginning element of the range of elements to be sorted by coding the keyword USING followed by an arithmetic expression. If a beginning element is so specified, an ending element must also be given by an arithmetic expression following the keyword TO.

The user may choose the beginning position of the field in each element which forms the basis of the sort by coding a semi-colon followed by an arithmetic expression. If a beginning position is so specified, an ending position must also be given by an arithmetic expression following a comma. If a range of elements was not selected by the user (see preceding paragraph), the keyword USING must precede the semi-colon.

Secondary considerations: (1) The sort is done in ascending ATASCII order. (2) If the length of an element is less than the ending position of the field being used as the basis of the sort, the field shall be shortened accordingly. This condition applies regardless of whether the field is specified implicitly or explicitly. (Note that if two compared fields are equal except that one is longer than the other, the longer one is greater than the shorter one. This is intuitively correct as well as being consistent with string comparisons made with other BASIC XL statements and operations.)

9.3.4.2 SORTDOWN

```
Format:   SORTDOWN savar [USING[aexp TO aexp] [;aexp,aexp] ]

Examples: SORTDOWN Stringarray$
          SORTDOWN Array$ USING Min TO Max
          SORTDOWN X$ ; 1,4
          SORTDOWN X$ Using 5 To 1Ø ; 4,8
```

This statement will sort selected elements of a specified string array in descending order, based on the contents of a selected portion (a "field") of each element of the array. Unless otherwise specified by the user, the field of each element which forms the basis for the sort shall consist of the entirety of each element. Unless otherwise specified by the user, all elements of the array will be selected to be sorted.

The user may choose the beginning element of the range of elements to be sorted by coding the keyword USING followed by an arithmetic expression. If a beginning element is so specified, an ending element must also be given by an arithmetic expression following the Keyword TO.

The user may choose the beginning position of the field in each element which forms the basis of the sort by coding a semi-colon followed by an arithmetic expression. If a beginning position is so specified, an ending position must also be given by an arithmetic expression following a comma. If a range of elements was not selected by the user (see preceding paragraph), the keyword USING must precede the semi-colon.

Secondary considerations: (1) The sort is done in descending ATASCII order. (2) If the length of an element is less than the ending position of the field being used as the basis of the sort, the field shall be shortened accordingly. This condition applies regardless of whether the field is specified implicitly or explicitly. (Note that if two compared fields are equal except that one is longer than the other, the longer one is greater than the shorter one. This is intuitively correct as well as being consistent with string comparisons made with other BASIC XL statements and operations.)

9.4 Example BASIC XL Programs with Extended Statements

This section gives examples of programs written using the extended statements presented in section 9.3. Three of the programs here (those in sections 9.4.1, 9.4.2, and 9.4.3) are "brand new", presenting aspects of the extended statements which are very difficult to duplicate in BASIC XL (or any BASIC) without the unique capabilities of the extended statements. Of necessity, then, their descriptions are somewhat detailed.

The other three programs are retreads of three of our old friends from section 2. We present them again here to show you how you can turn a hard-to-read program riddled with GOSUBs into a well structured exercise. For these programs, only the significant differences from their originals are noted. You are invited to peruse the descriptions in section 9.2 for details on other parts of these programs.

9.4.1 FACTOR.BXE

For such a short program, this will be a rather long explanation. The program given here is actually one of the classic ones used to show how recursion works: We calculate the factorial of a number by repetitive calls to a procedure.

Now, actually, this is a fairly inefficient way to calculate a factorial. Perhaps the simplest way is the following little program:

```
10 Input "Give me a positive integer> ",N
20 P=1
30 For I=1 To N: P=P*I: Next I
40 Print N; "! is "; P
```

So if all you want is the factorial of a number, use the above routine and forget about the demo on the disk. But if you want to understand how recursion works, read on.

If you will examine a listing of FACTOR.BXE, you will find the first part, lines 100 through 220, rather ordinary and mundane. The possible sole exception is the CALL to the Factorial procedure, where we pass in a number and expect a result.

But now look at the Factorial procedure itself. If you recall our discussion of procedure parameters and local variables in section 9.3.3, you probably aren't too surprised to find the name used in the main routine reused here in the procedure. Recall also that the

effect of using an arithmetic variable either as a parameter (i.e., Number in this example) or as a local variable (i.e., Result) is that, upon Exit from the Procedure, its original value is restored. Now, there isn't really any reason to use these same variable names again in this program other than as a teaching mechanism, but its a fairly effective mechanism.

Well, once we get past the Procedure and local declarations, there isn't much left to the routine, so let's examine it in close detail.

Since the main code ensured that we would, indeed, use a positive integer for Number, we know that we have a number which will produce a valid factorial. Now, the factorial of 1 is 1, so line 28Ø makes sense: If the parameter is 1, then Exit with an answer of 1. Simple. Clean. Neat.

Just as an exercise, let's assume that we want the factorial of 3. Okay, Number is not 1, so we get to line 29Ø. How about that? We turn around and Call ourselves again, but this time our calling parameter has a value of 2 (...Using Number-1...). Let's keep going.

We're back at line 28Ø. But Number now has a value of 2, so we don't take the Exit here. Instead, we once again Call ourselves. Ready to keep going?

Back at line 28Ø, Number now has a value of 1. Aha! Finally, we get to Exit with a value of 1. But wait a minute? Certainly 3! is not 1, is it? Not to worry. Remember, the last time we called the procedure, we did so from line 29Ø, when Number had a value of 2. Okay, so we return back to that same line 29Ø, and Result gets a value of 1. Then we continue on to line 3ØØ, where we Exit with what?

Well, we just said Result is 1, and since Number had a value of 2 when line 29Ø made the call, that value has been restored by now (as we noted above). So Number*Result is 2*1, and we Exit with a value of 2.

But where do we Exit back to? Well, we got rid of the last of the calls on that last Exit, so this time we end up back at line 29Ø from the time we called with Number equal to 3, and Result gets a value of 2. By the same logic, we continue to line 3ØØ and Exit with 3*2.

This time, though, we have dispensed with all the calls except the original one, in line 19Ø, so that Result gets the Exit value of 3*2, or 6. Voila! 3! is truly 6, as we wanted.

There was nothing magic about our choice of 3 for our example. The principle holds no matter what the value we use: Keep calling the procedure with successively smaller values until the value reaches 1. Then start Exiting back up the Call chain, multiplying as we go. Terribly inefficient, but a beautiful example of classical recursion at work.

So, do you see the advantage of truly local values, not only for parameters but for other explicitly declared variables? No? You think this was an artificially created example? Well, just wait ... we have some more realistic examples coming up.

Technical Sidelight: By the way, try to discover the largest integer whose factorial can be represented within your Atari's numeric range (it's less than 1ØØ). Then try finding out what 1ØØ! is. Bang! You got numeric overflow when the multiplies created a result larger than Atari floating point can represent. But for real fun, try finding out what 5ØØØ! is. Do you understand why you got that error? Does it help if we remind you that each local or parameter variable uses 12 bytes of memory? And that each Call itself uses 4 bytes? Hmmm ... how much memory does your machine have? (To get rid of all that junk on the stack, just use the CLR command from the Ready prompt level.)

9.4.2 SORTDIR.BXE

This isn't really a very exciting program. All it does is read in a disk directory and then allow you to choose which one of three ways you would like to see it sorted. Its primary purpose is to show how you may sort on different "fields" within the single "record" each element of a a string array can represent.

1ØØ-24Ø Just the usual necessary set up. Note the names given to the console keys; obviously not a necessary step, but one which makes a prettier program. The FILES() array is dimensioned large enough to hold the largest directory a standard DOS 2 disk will allow. If your DOS allows more files, or if the entries in the directory are longer, feel free to change the DIMensions.

26Ø, 6ØØ By now, you are used to seeing endless WHILE loops in our programs. The beginning of this loop may be in the wrong place for you. As is, it reads the directory in off the disk each time a new sort is done. This is so that you can change diskettes if you wish. It might have been better to at least give you a chance to tell the program that you have changed disks. Sounds like a good programming exercise for you to us.

270-340 This is an easy way to read in the directory. The LINES variable is not really needed -- you can INPUT directly into a string array element if you wish -- but it avoids having the "FREE SECTORS" line end up in the array. Just a small nicety.

Notice how we depend on the space in the second character position for each directory line except the "xxx FREE SECTORS" of the final line.

350-390 Self-explanatory. Actually, we could have special cased a directory with a single file (why bother to sort it?), but it isn't necessary.

400-480 After presenting the menu, a beep (PUT #0,253) reminds you to push a button. After you do, we clear the screen.

490-560 This is what we really wanted to demonstrate. Depending on which button you pushed, we SORTUP based on a particular field. The SORTUP statements of lines 500, 520, and 540 are identical except for the numbers following the semi-colon. Inspect a single line of the directory listing. Do you see how the numbers are the character positions within the line? Easy, isn't it.

Notice, also, that we do not sort the entire array. Rather, we only sort the part which holds valid directory entries. Also very easy, right?

588-648 Just a way to display the directory in two columns. The sorted listing reads down the first column and then down the second. It would have been easier to simply alternate, but this is easier to scan visually.

Again, feel free to modify this program to your liking.

9.4.3 SORTNUM.BXE

In the presentation of the sort statements in section 9.3.3, we discussed a way to sort integers by converting them into a consistent form in a string. This program presents a different and more general way to sort the floating point numbers which BASIC XL (and Atari BASIC) uses.

Performing this sort depends upon knowing the internal format of floating point numbers used by BASIC. The form is fairly simple: A single byte of sign and exponent followed by 10 BCD digits, two to a byte. The sign of the number is given by the uppermost bit of that first byte. The exponent is a power of 100 in what

is known as "excess-64" form. (That means that the true power of 1ØØ has 64 added to it so that all exponents appear as positive numbers. To form the true exponent, then, subtract 64 from the byte after getting rid of the sign bit.)

If you study this format, you will discover a fortuitous occurrence: If you treat the six bytes of a positive number as if they were a string, positive numbers will automatically be sorted correctly by SORT UP and SORTDOWN. Truthfully, this is not a coincidence. Internal to BASIC, such consistency is used for comparisons (e.g., as when you code something like IF A>B THEN ...).

On the other hand, because negative numbers have that upper bit set, they will all sort as larger than any positive number! Oops, to say the least. Not only that, if you ignore the sign bit, the negative numbers look exactly like positive numbers, so they will be sorted in reverse order. And, finally, what about zero. Which consists of six bytes of $ØØ? Well, it is now time to examine the program listing to see how we turned adversity to advantage.

15Ø-16Ø The only reason for the DUMMY$ string is to provide an address for that single element numeric array. Recall that in BASIC XL (and Atari BASIC), string and array variables always use memory in the order they are DIMensioned. Thus the address of VALUE has to be one greater than the address of DUMMY$.

18Ø This array is actually going to hold our array of floating point numbers. In fact, notice that it is the same size as an array of 2Ø numbers. Of course we have to use a string array because SORTUP And SORTDOWN can only handle string arrays. That's only a minor inconvenience, as we shall see.

28Ø, 36Ø We're going to generate, manipulate, and display 2Ø random numbers.

29Ø This is just to give each element of the array a LENgth of six. Otherwise, the sort process won't know how many bytes in each array element need sorting.

3ØØ We generate random numbers in an arbitrary range, but one which is easy to view.

31Ø-32Ø See how we move the six bytes of the floating pointer number into the element of the string array? Didn't know you could do that in BASIC?

330 All we do here is flip the state of the sign bit: if the number was positive, it is now negative; and vice versa. Note the effect of this: What were negative numbers will now sort as smaller than what what were positive numbers. Just think of that bit as representing a plus sign now, instead of a minus sign.

340 We count all the numbers which were negative. Don't worry why. We'll show you.

350 We Just display the numbers in an easy to view form. Mixed up bunch of digits, aren't they?

370-380, 410-420 The only reason for these lines is so that you can see how fast the array is sorted. Pretty impressive, even if it is only 26 numbers. Feel free to try it with more.

390 Okay. This is obvious. Everything is now sorted very prettily. Except that playing games with that sign bit didn't fix the fact that the negative numbers will be sorted backwards.

400 The magic. Because we kept track of the count of negative numbers, and because the SORTUP of line 390 put all the negative numbers before the positive ones in the array, this works! We simply re-sort the negative numbers in backward order via SORTDOWN. You'll simply have to RUN this program to believe it.

440-490 This loop just displays the now sorted array. Note how we now have to flip the sign bit back to its original state before moving it back to VALUE(0) for printing. Not very hard, right? (Actually, we didn't have to flip the bit. We could have moved the number as is and then printed -VALUE(0) for the same effect. But the way shown is more orderly.)

That's it. The best part of this method is that you could easily incorporate the six byte 'field' of the floating point number into a longer "record" so that you could sort the array several ways, as we did in the last section.

9.4.4 GTIATEST.BXE

This is the first of our "conversions' from a standard BASIC XL version to one using extended statements. In the mainline code, line 1040 has been changed to a CALL. The subroutine starting at line 9000 has been turned into a PROCEDURE, and the variables used in it have been made LOCAL (line 9080).

Now, truthfully, there was little incentive to change this routine into a Procedure. What have we saved? The variables are local, so they can get used for other purposes elsewhere in the program. And since we Exit with the test value, the caller doesn't have to aware of name we use in the subroutine. Big deal.

No, the real reason we changed this program was once again instructional. We just wanted to show how easy it really is to use Procedures and write readable code. There's more to come.

9.4.5 DISKIO.BXE

Another fairly simple conversion from the original standard BASIC XL program . This time, though, there is little more justification for using Procedures.

Just look at lines 9560, 9600, 9620, and 9660. What could be clearer? Just think: You could have an entire library of Procedures sitting around on disks. And you could keep a listing of just the entry (Procedure) and Exit lines. You almost wouldn't need any other documentation, would you?

Watch how easy it is to use these routines if the code from 9000 up is included in your code:

```
10 Dim High$(128) : High$="0000000000"
20 Call "Read Sector" Using 1,720,Adr(High$),1 To Test
30 Print "High score is ";Val(High$)
40 Input "New high score ? ",High
50 High$=Str$(High),Chr$($9B)
60 Call "Write Sector" Using 1,720,Adr(High$),1 To Test
70 Stop
```

If you included something like that in your code, you could save the high score from a game in the usually invisible sector 720. Cute?

Trickies in that code: We give High$ that initial value so that it will have a valid LENgth (like BGET, direct sector access doesn't change the length of a string).

Similarly, we put a RETURN character into the string (line 5Ø) so that a later sector read and VAL() will find something to terminate the number.

Finally, we leave you with the thought that a sector holds 128 bytes. If you used a string array such as

```
     DIM High$(11,1Ø)
```

and then, in the Call used ADR(High$(1;))-2 (minus 2 so that we get the length bytes for the first element of the array), we could keep track of up to 1Ø high scores with, perhaps, 3 initials and up to 7 digits of score each. (Why not 11 scores, when we dimensioned the array to have 11 elements? Well, the actual size of that array in bytes is 11*(1Ø+2) or 132 bytes, where the +2 accounts for the length bytes in each element. But the sector can only hold 128 bytes, so we would be missing 4 bytes from the last element.)

9.4.6 PHONE.BXE

This last program "conversion" is our "Little Black Book" program from section 9.2.9. It was a monster as a standard program. It remains a monster using extended statements. But, perhaps, it is a more manageable monster now.

Actually, we changed the character of the program very little. And we even tried to keep all subroutines at or near the same line numbers. What we tried to do was change every GOSUB to a CALL. Now, we will admit that some of the routines didn't really need to be made into Procedures, but once again it is at worst an educational exercise.

We invite you to peruse especially the Procedures in lines 5ØØØ through 9999. What you might find most interesting is looking for the variables which we left global, those we did not pass as parameters. The most notable of these are strings used as field names (e.g., Last$) and file names (DBX$, DBF$). The hassle of making these into parameters every place they are used was fueled with the likelihood that in any application of this system you would most likely use only one data base file at a time. Result: they are left global.

On the other hand, look at the "Get Line" routine, lines 5ØØØ to 526Ø. Here was a great opportunity to pass a string both in and out, thus allowing us to put the edited line directly into the user's string variable space, no muss, no fuss. This same Procedure benefits by being able to easily call it with the maximum number of characters you want to get as well as a flag determining the fate of lower case letters.

And look at all the routines which use the variables Temp1 and Temp2, which they inevitably make into LOCAL variables. How nice it is to not have to worry about possible conflicts in temporary variable usage anymore.

Similarly, "Make Index" starting at line 7500 shows off its usage of parameters passed to it. Look at the Call to it in line 20240. How nice to not be forced into making variable names match!

Aside from all of that, you might look at the code in lines 1570 through 1710. Notice how we build up two string arrays with the names of our Procedures carefully ensconced as elements therein. Then look at line 2260 and lines 10250 and 10260. Do you see how we can use a menu option to nicely choose even the correct Procedure to call?

The most important aspect of all this, though, may be that now the routines have been somewhat freed of the tyranny of line numbers and variable names. Feel free to copy them and use them in your own programs. Who knows? You may be a budding data base programmer who just hasn't had the right tools. Until now.

Appendix A: ERROR DESCRIPTIONS

ERROR NUMBER	DESCRIPTION

1 While SET Ø,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 Ø to 65535 (hex $FFFF) is called for and the given expression is either too large or negative.
>
> ```
> A = PEEK(-1)
> DIM B(7ØØØØ)
> ```
>
> Both these statements will produce a value error.

Example:

> An expression that can be converted to a one byte integer in the range ~ to 255 hex(FF) is called for and the given expression is too large.
>
> ```
> POKE 5ØØØ,75Ø
> ```
>
> This statement produces a value error.

Example:

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

4 No more variables can be defined. The maximum number of variables is 128.

ERROR NUMBER	DESCRIPTION

5 A character beyond the DIMensioned or current length of a string has been accessed. Example:

```
1ØØØ DIM A$(3)
2ØØØ A$(5) = "A"
```

This will produce a string length error at line 2ØØØ 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.

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

```
1ØØØ READ A
2ØØØ PRINT A
3ØØØ END
4ØØØ DATA 12AB
```

Running this program will produce this error.

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

```
1ØØØ DIM A(1Ø)
2ØØØ DIM A(1Ø)
```

This program produces a DIM error.

1Ø 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:

```
1ØØØ IF A=B THEN 5ØØ
```

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

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

```
1ØØØ DIM A(1Ø)
2ØØØ REM FILL THE ARRAY
3ØØØ FOR I = Ø TO 1Ø
4ØØØ A(I) = I
5ØØØ NEXT I
6ØØØ REM PRINT THE ARRAY
7ØØØ FOR K = Ø TO 1Ø
8ØØØ PRINT A(K)
9ØØØ NEXT K
1ØØØØ END
```

Running this program will cause the following output:

```
Ø
ERROR- 13 AT LINE 9ØØØ
```

NOTE: Improper use of POP could cause this error.

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:

```
1ØØØ GOSUB 2ØØØ
11ØØ PRINT "RETURNED FROM SUB"
12ØØ END
2ØØØ PRINT "GOT TO SUB"
21ØØ STOP
22ØØ RETURN
```

If this program is run the print out is:

```
GOT TO SUB
STOPPED    AT LINE 21ØØ
```

ERROR NUMBER | DESCRIPTION

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.

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

ERROR
NUMBER DESCRIPTION

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 6Ø.

23 The value of a variable in a PRINT USING statement is greater than or equal to 1E+5Ø.

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.

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:

```
1Ø WHILE 1 : REM loop forever
2Ø GOSUB 1ØØ
1ØØ ENDWHILE
```

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

ERROR NUMBER	DESCRIPTION
29	An illegal player/missile number. Players must be numbered from Ø-3 and missiles from 4-7.
3Ø	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 Ø is invalid.
35	When RENUMbering, the maximum line (32767) was exceeded.
4Ø	You attempted to use a string variable as a string array variable, or visa versa. Example: DIM A$(3,29) A$="THIS CAUSES AN ERROR" would create this error.

LABEL	HEXADECIMAL LOCATION	COMMENTS and DESCRIPTION
APPMHI	DE	Highest location used by BASIC XL (LSB, MSB)
RTCLOK	12,13,14	Screen Frame Counter (1/6Ø sec.) (LSB, NSB, MSB)
SOUNDR	41	Noisy I/O Flag (Ø=quiet)
ATRACT	4D	Attract Mode Flag (128=Attract Mode)
LMRGIN, RMRGIN	52,53	Left, Right Margin (Defaults 2, 39)
RAMTOP	6A	Actual top of memory (page number)
LOMEM	8Ø,81	BASIC XL low memory pointer
MEMTOP	9Ø,91	BASIC XL high memory pointer (usually same as APPMHI)
FRØ	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	2FØ	Cursor Inhibit (Ø=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)

LABEL	HEXADECIMAL LOCATION	COMMENTS and DESCRIPTION
CONSOL	DØ1F	Console Keys (bit 2=OPTION; bit 1=SELECT; bit Ø=START)
SKCTL	D2ØF	Serial Port Control Register (bit 2=Ø if last key still pressed)

```
     $ØØ  +----------------------------+
          |       OS Variables         |
     $8Ø  +----------------------------+
          |   BASIC XL System RAM      |
     $CB  +----------------------------+
          |    Free BASIC XL RAM       |
     $D2  +----------------------------+
          |  ATARI Floating Point      |
          |       Registers            |
    $1ØØ  +----------------------------+
          |     Hardware Stack         |
    $2ØØ  +----------------------------+
          |       OS Variables         |
          |          IOCBs             |
    $3CØ  +----------------------------+
          |     Printer Buffer         |
    $3E8  +----------------------------+
          |         OS RAM             |
    $3FD  +----------------------------+
          |     Cassette Buffer        |
    $48Ø  +----------------------------+
          |   BASIC XL Stack and       |
          | Miscellaneous Variables    |
    $57E  +----------------------------+
          |   ACTION! Hash Tables      |
    $68Ø  +----------------------------+
          |        Free RAM            |
    $7ØØ  +----------------------------+
          |         DOS RAM            |
 (MEMLO)  +----------------------------+
          | BASIC XL program, buffers, |
          |  tables, run-time stack.   |
(APPMHI)  +----------------------------+
          |        Free RAM            |
  MEMTOP  +----------------------------+
          |      Screen Memory         |
          |  also optional P/M Memory  |
   $AØØØ  +----------------------------+
          |   BASIC XL Cartridge       |
   $CØØØ  +----------------------------+
          |      OS, ROMs, etc.        |
   $DØØØ  +----------------------------+
          |   Hardware Registers       |
   $D8ØØ  +----------------------------+
          |  OS and Floating Pt. ROM   |
   $FFFF  +----------------------------+
```

ATASCII stands for "ATARI ASCII". Letters and numbers have the same values as those in ASCII, others are different.

Dec	Hex	CHARACTER	Dec	Hex	CHARACTER	DEC	HEX	CHARACTER
0	$00	♥ á	32	$20	█	64	$40	@
1	$01	├ ù	33	$21	!	65	$41	A
2	$02	▕ Ñ	34	$22	"	66	$42	B
3	$03	┘ É	35	$23	#	67	$43	C
4	$04	┤ ç	36	$24	$	68	$44	D
5	$05	┐ ô	37	$25	%	69	$45	E
6	$06	╱ ò	38	$26	&	70	$46	F
7	$07	╲ ì	39	$27	'	71	$47	G
8	$08	◢ £	40	$28	(	72	$48	H
9	$09	▗ ï	41	$29	)	73	$49	I
10	$0A	◣ ü	42	$2A	*	74	$4A	J
11	$0B	▝ ä	43	$2B	+	75	$4B	K
12	$0C	▘ Ö	44	$2C	,	76	$4C	L
13	$0D	▔ ú	45	$2D	-	77	$4D	M
14	$0E	▁ ó	46	$2E	.	78	$4E	N
15	$0F	▖ ö	47	$2F	/	79	$4F	O
16	$10	♣ Ü	48	$30	0	80	$50	P
17	$11	┌ â	49	$31	1	81	$51	Q
18	$12	─ û	50	$32	2	82	$52	R
19	$13	┼ î	51	$33	3	83	$53	S
20	$14	● é	52	$34	4	84	$54	T
21	$15	▄ è	53	$35	5	85	$55	U
22	$16	▎ ñ	54	$36	6	86	$56	V
23	$17	┬ ê	55	$37	7	87	$57	W
24	$18	┴ å	56	$38	8	88	$58	X
25	$19	▌ à	57	$39	9	89	$59	Y
26	$1A	└ Å	58	$3A	:	90	$5A	Z
27	$1B	␛	59	$3B	;	91	$5B	[
28	$1C	↑	60	$3C	<	92	$5C	\
29	$1D	↓	61	$3D	=	93	$5D	]
30	$1E	←	62	$3E	>	94	$5E	^
31	$1F	→	63	$3F	?	95	$5F	_

Note: Columns with two symbols for one dec/hex value show standard ATASCII to the left, and to the right the symbol from the international character set available with XL/XE computers.

Dec	Hex	CHARACTER	Dec	Hex	CHARACTER	DEC	HEX	CHARACTER
96	$60	♦	128	$80		160	$A0	
97	$61	a	129	$81		161	$A1	
98	$62	b	130	$82		162	$A2	
99	$63	c	131	$83		163	$A3	
100	$64	d	132	$84		164	$A4	
101	$65	e	133	$85		165	$A5	
102	$66	f	134	$86		166	$A6	
103	$67	g	135	$87		167	$A7	
104	$68	h	136	$88		168	$A8	
105	$69	i	137	$89		169	$A9	
106	$6A	j	138	$8A		170	$AA	
107	$6B	k	139	$8B		171	$AB	
108	$6C	l	140	$8C		172	$AC	
109	$6D	m	141	$8D		173	$AD	
110	$6E	n	142	$8E		174	$AE	
111	$6F	o	143	$8F		175	$AF	
112	$70	p	144	$90		176	$B0	
113	$71	q	145	$91		177	$B1	
114	$72	r	146	$92		178	$B2	
115	$73	s	147	$93		179	$B3	
116	$74	t	148	$94		180	$B4	
117	$75	u	149	$95		181	$B5	
118	$76	v	150	$96		182	$B6	
119	$77	w	151	$97		183	$B7	
120	$78	x	152	$98		184	$B8	
121	$79	y	153	$99		185	$B9	
122	$7A	z	154	$9A		186	$BA	
123	$7B	♣	155	$9B		187	$BB	
124	$7C	|	156	$9C		188	$BC	
125	$7D		157	$9D		189	$BD	
126	$7E	◀	158	$9E		190	$BE	
127	$7F	▶	159	$9F		191	$BF	

Notes: Add 32 to upper case code to get lower case code for same letter.

Characters from 128 to 255 are reverse colors of 1 to 127.

Dec	Hex	CHARACTER	Dec	Hex	CHARACTER
192	$CØ	@	224	$EØ	◆ i
193	$C1	A	225	$E1	a
194	$C2	B	226	$E2	b
195	$C3	C	227	$E3	c
196	$C4	D	228	$E4	d
197	$C5	E	229	$E5	e
198	$C6	F	23Ø	$E6	f
199	$C7	G	231	$E7	g
2ØØ	$C8	H	232	$E8	h
2Ø1	$C9	I	233	$E9	i
2Ø2	$CA	J	234	$EA	j
2Ø3	$CB	K	235	$EB	k
2Ø4	$CC	L	236	$EC	l
2Ø5	$CD	M	237	$ED	m
2Ø6	$CE	N	238	$EE	n
2Ø7	$CF	O	239	$EF	o
2Ø8	$DØ	P	24Ø	$FØ	p
2Ø9	$D1	Q	241	$F1	q
21Ø	$D2	R	242	$F2	r
211	$D3	S	243	$F3	s
212	$D4	T	244	$F4	t
213	$D5	U	245	$F5	u
214	$D6	V	246	$F6	v
215	$D7	W	247	$F7	w
216	$D8	X	248	$F8	x
217	$D9	Y	249	$F9	y
218	$DA	Z	25Ø	$FA	z
219	$DB	[	251	$FB	♠
22Ø	$DC	\	252	$FC	|
221	$DD	]	253	$FD	↰
222	$DE	^	254	$FE	◀
223	$DF	_	255	$FF	▶

Note: To get ATASCII code, tell computer (direct mode) to PRINT ASC ("__"). Fill blank with letter, character, or number of code. Must use the quotes!

Appendix E: SYNTAX SUMMARY AND KEYWORD INDEX

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
aexp	-	arithmetic exp
sexp	-	string exp
var	—	variable
avar	-	arithmetic var
svar	-	string var
mvar	-	matrix var (or element)
fn	-	file number
<stmts>	-	one or more statements
filename	-	svar or string literal (quotes are optional except with LIST)
line	-	line number (can be aexp)
pm	-	Player/Missile number (aexp)
[xxx]	-	xxx is optional
[xxx...]	-	xxx is optional, and may be repeated
addr	-	address aexp, must be 0 - 65535

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

STATEMENTS

page	syntax
49	*BGET #fn,addr,len
50	*BPUT #fn,addr,len
21	BYE
50	CLOAD
51	CLOSE #fn
21	CLR
91	COLOR aexp
22	CONT
23	*CP
51	CSAVE
51	DATA <ATASCII data>
72	DEG
22	*DEL line[,line]

```
page   syntax
----   ------
10     DIM avar(aexp)
10     DIM mvar(aexp,aexp])
12     DIM svar(aexp)
12     *DIM savar(aexp, aexp)
52     *DIR [filename]
23     DOS
82     *DPOKE addr,aexp
92     DRAWTO aexp,aexp
40     *ELSE {see IF}
34     END
40     *ENDIF {see IF}
46     *ENDWHILE
52     ENTER filename
53     *ERASE filename
23     *FAST
35     FOR avar=aexp TO aexp [STEP aexp]
53     GET #fn,avar
36     GOSUB line
37     GOTO line
87     GRAPHICS aexp
39     IF aexp THEN <stmts>
39     IF aexp THEN line
40     *IF aexp : <stmts>
           ELSE : <stmts>
           ENDIF
54     *INPUT "...",var [,var...]
53     INPUT [#fn,] var [,var...]
41     *[LET] svar=sexp [,sexp...]
41     [LET] avar=aexp
41     [LET] mvar=aexp
24     LIST [filename]
24     LIST [filename,] line [,line]
55     LOAD filename
92     LOCATE aexp,aexp,avar
24     *LOMEM addr
55     LPRINT [exp [;exp...] [,exp...] ]
25     *LVAR [filename]
105    *MISSILE pm,aexp,aexp
43     *MOVE fromaddr,toaddr,lenaexp
25     NEW
35     NEXT avar
55     NOTE #fn,avar,avar
25     *NUM [line] [,aexp]
43     ON aexp GOTO line [,line...]
43     ON aexp GOSUB line [,line...]
56     OPEN #fn,mode,avar,filename
93     PLOT aexp,aexp
102    *PMCLR pm
```

```
102     *PMCOLOR pm,aexp,aexp
102     *PMGRAPHICS aexp
104     *PMMOVE pm[,aexp] [;aexp]
page    syntax
----    ------
105     *PMWIDTH pm,aexp
57      POINT #fn,avar,avar
83      POKE addr,aexp
44      POP
93      POSITION aexp,aexp
57      PRINT [#fn]
57      PRINT exp [ [;exp...] [,exp...] ] [;]
57      PRINT #fn [ [;exp...] [,exp...] ] [;]
58      *PRINT [#fn,] USING sexp, [exp [,exp...] ]
67      *PROTECT filename
63      PUT #fn,aexp
72      RAD
70      RANDOM
63      READ var [,var...]
26      REM <any remark>
64      *RENAME filenames
27      *RENUM [start][,increment]
45      RESTORE [line]
36      RETURN
64      *RGET #fn, svar [,svar...]
64      *RGET #fn, avar [,avar...]
65      *RPUT #fn,exp [,exp...]
27      RUN [filename]
66      SAVE filename
28      *SET aexp,aexp
94      SETCOLOR aexp,aexp,aexp
97      SOUND aexp,aexp,aexp,aexp
66      STATUS #fn,avar
35      STEP {see FOR}
31      STOP
67      *TAB [#fn], avar
39      THEN {see IF}
35      TO {see FOR}
31      *TRACE
31      *TRACEOFF
45      TRAP line
67      *UNPROTECT filename
46      *WHILE aexp
67      XIO aexp,#fn,aexp,aexp,filename
57      ? {same as PRINT}
```

FUNCTIONS
----

page	syntax
69	ABS(aexp)
81	ADR(svar)
73	ASC(sexp)
72	ATN(aexp)
8Ø	*BUMP (pmnum,aexp)
73	CHR$(aexp)
69	CLOG(aexp)
72	COS(aexp)
81	*DPEEK(addr)
82	*ERR(aexp)
7Ø	EXP(aexp)
74	*FIND(sexp,sexp,aexp)
82	FRE(Ø)
78	*HSTICK(aexp)
7Ø	INT(aexp)
75	LEN(sexp)
7Ø	LOG (aexp)
78	PADDLE (aexp)
78	*PEN(aexp)
81	*PMADR(pm)
78	PTRIG(aexp)
83	PEEK(addr)
71	RND(Ø)
71	SGN(aexp)
72	SIN(aexp)
71	SQR(aexp)
79	STICK(aexp)
79	STRIG(aexp)
76	STR$(aexp)
84	*SYS(aexp)
84	*TAB(aexp)
84	USR(addr [,aexp...])
76	VAL(sexp)
79	*VSTICK(aexp)

Appendix F: COMPATIBILITY WITH ATARI BASIC

Generally, BASIC XL is totally compatible with Atari BASIC. Virtually all programs written in Atari BASIC and SAVEd or CSAVEd thereunder 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 Ø
```

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 as 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,Ø
```

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. Programs Which RUN Too Fast

Of course, 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.

4. Memory Locations

BASIC XL attempts to conform 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 4ØØ/8ØØ,
     by Atari, Inc.
        (except that locations SIN, COS, ATAN, and SQR
        are incorrect, even for Atari BASIC)

The BASIC XL Programming Environment

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.

```
$8Ø 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 DIMENSION

BASIC XL automatically dimensions strings to 4Ø characters. Again, this should have no effect on currently running Atari BASIC programs. If desired, you can use

```
SET 11,9
```

to ensure total 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 then re-ENTERed into BASIC. Again, you may use

```
SET 12,9
```

to ensure compatibility and remove the indenting.

These are the benchmark tests known from Compute! (issue 57, February 1985, pp. 139-142), adapted to BASIC XL.

INSIGHT: Atari by Bill Wilkinson

I am much gratified by the response to my decision to work harder on answering readers' questions. I have received several very interesting letters with both good comments and good questions. Since it is always fun to defend Atari BASIC against the outside world, let me start with a subject near and dear to my heart.

Benchmarks

Several readers have asked me why Atari BASIC compares so unfavorably to other computers on certain benchmarks. The two most commonly mentioned are the BYTE magazine benchmarks (September, 1981 BYTE Magazine, pp. 180-198) and the Creative Computing benchmark invented by David Ahl (November, 1983 Creative Computing Magazine, pp. 259-260). Stan Smith, of Los Angeles, asked some very pointed questions, which I will try to answer here.

The BYTE benchmark is reproduced below in Atari BASIC. It is the often-mentioned "Sieve of Erastothenes," a program which produces (and counts) prime numbers. Its primary advantage as a benchmark is that it can be implemented in virtually any language (although only with much difficulty when using Logo and its ilk). It relies only on addition and logical choices, with very little number crunching.

```
10 DIM N$(8192)
20 N$="0":N$(8192)="0":N$(2,8192)=N$
30 FOR I=1 TO 8192:IF N$(I,I)="1" THEN 60
40 PRIME=I+I+I:CNT=CNT+1:K=I
50 K=K+PRIME:IF K<8193 THEN N$(K,K)="1":GOTO 50
60 NEXT I
70 PRINT CNT : REM BETTER PRINT 1899!!!
```

An aside: If you have seen the BYTE original and are puzzled by my changes, be aware of three things: (1) I had to use a string because there is not enough room for an array of 8192 elements. (2) The math was modified very slightly to accommodate the fact that string indices start at one, instead of zero. (3) Multiple statements per line simplify the original somewhat.

Anyway, why is Atari BASIC so slow (317 seconds versus, for example, the IBM PC at 194 seconds)? Primarily for three reasons. First, note all the numbers in this listing, which must be treated as integers. Line numbers and indices are always kept and calculated as floating-point numbers, but all must be converted to integers before being used. (You simply can't GOTO line 137.38, can you?) And, sigh, the routine in the Atari Operating System ROMs which converts numbers to integers is incredibly slow (in fact, it is the only floating-point routine we modified when we produced BASIC A+ and BASIC XL).

Second, Atari BASIC performs FOR-NEXT loops by remembering the line number of the FOR statement. Then, when NEXT is encountered, BASIC must search for the FOR line, just as if a GOTO had been used. (Other BASICS remember the actual memory address of the FOR statement. Faster, but less flexible. Atari BASIC allows you to STOP in the middle of a loop, change the program, and continue, something no other home computer BASIC allows. (This - among many other things - is in direct opposition to Consumer Reports' claim that Atari BASIC is hard for beginners.)

Third, if you type in and use this listing as shown, you are paying almost a 5Ø percent penalty in speed, thanks to Atari's screen DMA and Vertical Blank Interrupts taking up a significant portion of the processing time. The simple addition of the following two lines will improve the time for this little test to 211 seconds:

```
5  POKE 54286,Ø : POKE 54272,Ø
65 POKE 54286,64
```

All of a sudden, Atari BASIC isn't even near the bottom of the list. And, yet, there is more we can do to improve the machine's performance. As many have suggested, you can install the Newell Fastchip, a replacement for the floating point routines built into your computer (available from many dealers, produced by Newell Industries of Plano, Texas).

Or you can change to another BASIC. Obviously, there is Atari's Microsoft BASIC. It produces results very close to those of Applesoft; but it, too, can be improved by turning off screen DMA, etc. And there is OSS's own BASIC XL. Using a combination of clever programming and a Fastchip, the BASIC XL program below will count up all those prime numbers in 58.5 seconds, about three times as fast as Microsoft BASIC on an IBM PC can do it. 'Nuff said. (Except a P.S.: The Set 3 in line 1Ø requests zero-time FOR loops, something not available in many BASICs, which alone accounts for about 2Ø seconds worth of improvement. See section 3.15 for details.)

```
10 FAST: POKE 54286,0: POKE 542
   72,0: SET 3,1: DIM N$(8192):
   N=ADR(N$)
30 FOR I=0 TO 8191
50 IF NOT PEEK(N+I) THEN PRIME=
   I+I+3:CNT=CNT+1:FOR K=I+PR
   IME TO 8191 STEP PRIME: POKE
   N+K,1: NEXT K
60 NEXT I
70 POKE 54286,64: POKE 559,34:
   PRINT CNT
```

Measures Of Accuracy

The Ahl benchmark is listed below. It purports to measure both accuracy and number-crunching ability. It does neither very well. Still, we have to ask why Atari BASIC is near dead last in its rankings, requiring 6 minutes and 45 seconds to complete the test.

```
10 FOR N=1 TO 100: A=N
20 FOR I=1 TO 10: A=SQR(A): R=R
   +RND(0): NEXT I
30 FOR I=1 TO 10: A=A^2: R=R+RN
   D(0): NEXT I
40 S=S+A: NEXT N
50 PRINT "ACCURACY="; ABS(1010-
   S/5), "RANDOM="; ABS(1000-R)
```

The culprit here (in terms of time-wasting) is line 30, with its A=A^2. Atari BASIC, in common with most small computer BASICS, calculates powers according to a formula:

```
x^y = exp(y * log(x))
```

where log() is the natural logarithm function and exp() is the exponent-of-e function.

If you don't understand that, don't worry about it. The point is that the calculation of such a simple thing as a number to the second power involves the calculation of a logarithm and an exponentiation. And why is that so bad? Simply because the floating-point routines in the Atari OS ROMs are too slow. Again, the solution is to install the Newell Fastchip and/or turn off DMA and VBI (as outlined above).

I am indebted to Clyde Spencer, one of the founders of the Bay Area Atari Users Group (one of the oldest), for supplying me with a most surprising figure. Spencer reports that, using the Fastchip and with DMA turned off, he obtained a timing of 1 minute 38 seconds, a

very respectable (albeit not record-shattering) performance. I still wouldn't use my Atari for advanced scientific applications, but it is more than adequate for most purposes.

There is a problem with the "accuracy" figures in this test, however. First, because Ahl's accuracy number is the result of 1000 simple sums, it is clearly possible that a particular machine may exhibit wildly variant results for various numbers and still show a good figure in his test. (To illustrate, assume that the SQR() function randomly tosses in an error of plus or minus one. If it tossed in an equal number of errors, they would balance to zero. Yet choosing to make the loop just one unit shorter [FOR N=1 TO 999] might give a completely different result. To be fair, this is a very unlikely result with modern math algorithms; but, still, one never knows.) A minor change to his program would improve the testing qualities considerably:

```
40 S=S+ABS(A-N):NEXT N
```

Do you see the difference? This method produces the sum of the errors, and doesn't fall prey to offsetting errors.

The Random Number Trap

There is no hope for the accuracy of this random number tester, though. I will quote Clyde Spencer on this matter: "If the numbers are truly random and not normally distributed, any difference between 0 and 1000 is possible. All you can say is that you would have a high probability of ... being near zero for a perfect random number generator." The benchmark test falls into the infamous BASIC repeating-random-sequence trap.

In most BASICS, when you command a program to run, the pseudo-random generator is always reseeded with the same number. So each and every time you will get the same results, with Ahl's test. And, depending on what seed is chosen, you may get truly phenomenal results (because you happened to hit a hot spot in the generator's sequence). Now, though, try starting the generator off with a different (and randomly chosen) seed each time. What happens? The test's randomness figure wanders all over the place.

Once again, to quote Spencer...... in eight tests I obtained numbers ranging from 1.6 to 24.2, with the mean being 7.02 ......

Finally, I would like to point out that Ahl's test penalizes small machine BASIC interpreters in yet another way: When you have 32K bytes to spend on a BASIC, one thing you do is insure that numbers to a

power are performed by successive multiplications, if possible. Thus Cromemco 32K Structured BASIC (for example) performs A^2 with just one multiply. In other words, it converts A^2 to A*A. If you manually substitute that same form in Ahl's program, the times for almost all of the smaller and less expensive machines will improve dramatically. (Surprisingly, though, the accuracy figures may not change. After all, the original version may have had offsetting errors.) Of course, if you need to use non-integer powers in your programs, this comment doesn't apply, and the benchmark's results are a bit more meaningful for you.

Well, what does all this long-winded discussion boil down to? Two simple points: (1) Always presume that a benchmark program is worth slightly less than the paper it is printed on. (2) If you want to do number crunching on your Atari computer (against my best advice), go out and buy the Newell Fastchip. (And it won't hurt to try some other languages.)

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 lines 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.

ToolKit: Comes with Runtime package. Now your programs go everywhere without a BASIC XL cartridge. And it contains new extended commands like: PROCEDURE, CALL, EXIT, LOCAL, and SORT. Assistance for techniques as Keyed File Access, Player/Missile Graphics, direct disk drive control, and much more.

BASIC XL

System

Run Time

Toolkit

GBXL
2015

A reference manual for

The BASIC XL Programming Environment

comprising

Cartridge Version
Run Time
Toolkit

A complete programming system designed
for your ATARI home computer system.

The programs, cartridges, ROMs, and manuals
comprising the BASIC XL environment
are Copyright (c) 1983, 1984 by
Optimized Systems Software, Inc.

Atari, Atari Computers, and Atari Home Computers are
trademarks of Atari, Inc.

This comprehensive information
provided by Atari enthusiasts
aims at the preservation of

The BASIC XL Programming Environment

1st revised edition (p) 2Ø22