A Reference Manual for the

BASIC XL TOOLKIT

Copyright (c) 1984, O.S.S., Inc.

Optimized Systems Software, Inc.
1221-B Kentwood Avenue
San Jose, California, 95129
Phone: (408) 446-3099

PREFACE

Congratulations on purchasing a copy of the **BASIC XL TOOLKIT.**

Before you begin your tour through this manual, we would like to call your attention to a couple of important matters.

RUNTIME BASIC XL

When you purchased this **BASIC XL TOOLKIT** package, you acquired the right to use the **RunTime** version of BASIC XL to distribute programs you have written in BASIC XL. You may distribute these programs on either a free basis (sometimes called "public domain") or on a commercial basis, for profit, without paying OSS any additional amounts.

However, before distributing a copy of the **RunTime** program, you must return a signed copy of the **License Agreement** included as part of this **ToolKit** package. The **License Agreement** provides, among other things, that you must affix a label bearing the copyright and trademark of O.S.S., Inc., to each and every copy which you distribute. Please read the **License Agreement** carefully for more details before signing and returning it to O.S.S., Inc.

BASIC XL Cartridge Versions

The extended BASIC XL statements described in Chapter 3 of this manual and the programs demonstrating the use thereof described in Chapter 4 **will not work** on BASIC XL cartridges with version numbers other than **1.02**! 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 1.02, you have two options:

(1) Return your BASIC XL ToolKit for a full refund. For our own peace of mind, you must also return your BASIC XL cartridge so that we may verify that it is indeed not version 1.02.

(2) You may purchase an updated version 1.02 cartridge from OSS for the postpaid price of $15, check or money order only, please. You must return your old BASIC XL cartridge with your check. (In dire circumstances, we can arrange to ship a cartridge and await receipt of yours. You must call us to arrange this, and you must have a MasterCard or Visa card handy. Extra charges will apply.)

Please note that current 1.02 cartridges **are** gold-plated (for longer and healthier life) and are beveled (for a better fit).

START PROGRAMMING!

Table of Contents

CHAPTER 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 witnout the BASIC XL cartridge.

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 <u>not</u> present on the disk, RunTime will continually try to find it. You should eject your diskette, shut off power, and try again.

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) <u>is</u> 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 2.1, for this purpose.

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 of 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.

1.4 Error Handling In RUNTIME BASIC XL

Errors which are **TRAP**ped by the running program are treated exactly the same way as when using the BASIC XL cartridge. Errors which are not **TRAP**ped 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.

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 load again **RUN** the program file, **AUTORUN.BXL**. If your partner **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.)

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.

CHAPTER 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 allow 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 thorough simply 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 5,0**).

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 1080 as

```
1080 Alpha = 64
```

but would that have any meaning to you? As we wrote it, the line clearly shows that **ALPHA** has a numeric value one less than the ATASCII value of the letter **A**.

1100 We chose the dimensions of **FILE$** 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 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 **PROTECT**ed 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 string array. Note the form it takes: "D:filename.BXL", where "filename" may have from 1 to 8 characters.

1320-1340 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 1340 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, **PRINT**ing **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.

1350-1370 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 1370. We are then done with the directory.

1410 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 1440). Sure, you can do it with **PEEK**s and **POKE**s 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**.)

1420 and 1510 This is an "endless" **WHILE** loop. We could have achieved the same thing by eliminating line 1420 and changing 1510 to read **GOTO 1430**. But that's terribly ugly! As well as being poor structured programming style.

1430-1470 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 01011111 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 1460, we validate that the letter chosen is in the range we have filenames for. (If it isn't, we skip to line 1500, 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...

1480-1490 Line 1480 illustrates the proper use of a **TRAP** in a well structured BASIC XL program. You should always **TRAP** to the next line of a loop or condition. Here, if we get an error in line 1490, we want to go back and ask for another menu selection. Voila. (Exception: Sometimes you will want to have a central routine for handling **TRAP**ped errors. That's a good idea, but beware of leaving **WHILE**s, **GOSUB**s, etc., sitting on the Run-Time 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 1470 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.

2.2 SNAILS

If you read **30 Days to Understanding BASIC XL** (or, better yet, worked your way through it), you will probably remember Chapter XXIX and an 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 2600 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 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.)

180 In the book, we had two variables (**SCORE0** and **SCORE1**) to keep track of the players' points. Here, we use a two element array. We'll show why below.

260 Isn't this easy to understand? You can translate this into English as follows: "As long as neither player has scored 10 points, keep playing!"

290 and 340 In the original, the COLORs are different. We changed them because it makes it easier to flash one of the slime trails (line 800).

490-500 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 690, 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 500 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 0000 0000 0011
          bitwise and with 0000 1111 1111
             binary result 0000 0000 0011
              (decimal value of 3)

MOVE = 243 decimal, binary 0000 1111 0011
          bitwise and with 0000 1111 1111
             binary result 0000 1111 0011
              (decimal value of 243)

MOVE = 258 decimal, binary 0001 0000 0010
          bitwise and with 0000 1111 1111
             binary result 0000 0000 0010
              (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 500 thus:

```
500 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:

```
500 Let Move=Move+3
690 Sound 0,Move&255,10,Volume
```

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

600-650 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.

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

```
760 If Bang0 <> Bang1
```

Because the values of **BANG0** and **BANG1** could be 1, 2, or 3, depending on who hit what. Using **NOT BANG0** 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 760 and all following references to **BANG0** and **BANG1** to this:

```
760 Bang0=Sgn(Bang0) : Bang1=Sgn(Bang1)
761 If Bang0<>Bang1
```

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

770 See line 760, 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 770:

```
770 If Bang1=0 : Bang(1)=Bang(1)+1
771 Else : Bang(0)=Bang(0)+1 : Endif
```

But, if you're willing to struggle with the logic a bit, you will conclude that our original line 770 achieves exactly the same result with less code.

800 Same thing again. Remember, **NOT BANG0** is a logical expression, so it can only take on numeric values of zero and one. Cute?

890 Another case of a logical expression being used to derive a numeric value. If **SCORE(0)** 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."

2.3 PICOADV

In addition to being the longest program on the ToolKit disk, **PICO-ADVENTURE** 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 <u>execution</u> 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.

100-119 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 120 column printer. If you are going to list this program to an 8 inch (nominal 80 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 <u>most</u> printers is to put its control or escape code sequence right into a **REM**ark 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 way 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. The 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 **PEEK**s and **POKE**s.

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, these counters are never allowed to become less than zero. One of them, **HUNGRY**, cycles from 20 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 **U**p and **D**own 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.

1520 and 2000-2120 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 **RETURN**s can then end up going back to the wrong place(s). We avoid the problem here by **GOSUB**bing to a known good line (2000).

Then, at line 2100, 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 17020. Suppose, though, that line 17020 doesn't exist (as it doesn't in **PICO**). Then the **TRAP 2110** is activated and we **GOTO** line 17000 instead.

Why? Well, as **PICO** is written, trying to **BURN MUSHROOM** will give us verb 7 and noun 2. Since line 17020 doesn't exist, we end up at line 17000, 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.

1600-1610 This **ELSE** clause was started by the **IF** of line 1510. The direction abbreviations (**N,E,S,W**) produce verb numbers of less than zero (-1 through -4). Once you understand the routine at line 7000, this part becomes easy.

7000-7050 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...

7100-7190 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
    100+ ... do a special action
```

The "special action" trick is a neat one, uniquely available only 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 use 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 <u>easy</u> 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 20.

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.** Line 13001 is pretty straightforward if you know how to read it: "If the Golem isn't carrying the requested object (if **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't 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.

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 10 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(0) in BASIC) and **RUN** the program from disk. You can play on two levels, beginner or advanced, but we recommend you 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 thrust 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 250 points for a great landing, 100 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 2500 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 **REM**arks and self-explanatory variable names). As with **PICO** (section 2.3) we will discuss this game in blocks, concentrating on the non-obvious features.

1000-1290 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 on 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 1180, and watch what happens.)

1300-1760 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 1700 to 1720, 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.

1800-1830 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 0. 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,(# 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 <u>not</u> 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 20, **0.96*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.

2600, 2770 This **WHILE** loop constitutes <u>all</u> 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.

2610-2620 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.

2630-2730 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 0.1 times the thrust, we use an intermediate variable (**LOSS**) to accumulate thrust in units of 10. When the **LOSS** exceeds 10, we use up a unit of fuel and reflect that fact in the fuel line on the left side (lines 2710 to 2730).

2740-2760 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.

2800-3060 For really great landings, we bring out the crowd. Note the way we assign the bonus points in line 3060.

3070-3250 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.

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 400 and 800 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, 10, 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 9000 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 0) 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.

9100 As long as we are testing, we might as well PRINT something which makes sense.

9130-9150 All of this ensures that we will place a black bar (player 0) right over the word **GTIA**.

9160-9210 We turn on the GTIA bits, wait for a clock tick, clear the collision registers, then wait at least two clock ticks.

9220 If $D004 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.

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 programs. Try it, it works.

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 starting 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 $6FF as our sector buffer. This is the infamous "page 6" which is so often overused. If you would like to avoid conflicts with other routines using page 6, feel free to locate the buffer anywhere else (e.g., within a **DIM**ensioned string). Second, note the way we print out the dumps. The **HEX$()** 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 **ESC**ape 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 **ESC**ape 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.

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 $0300 (768 decimal). Its contents are as follows:

Location	# of bytes	Description
$0300	1	Physical Device ID
$0301	1	Device Unit Number
$0302	1	Device Command Character
$0303	1	Data movement control (on call) SIO Returns Status (on exit)
$0304	2	Buffer Address
$0306	2	Timeout value
$0308	2	Buffer Length
$030A	2	Auxilliary 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 $01). 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 $303 has two uses. When you call **SIO**, it must contain $40 if you wish to obtain data from a device or $80 if you need to send data. A few device control commands need to neither read nor write data, so they use a value of $00 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 40 byte hunks. 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 auxilliary information is sent unmodified to the device, along with the command. Each device chooses what the auxilliary info implies, but for disk drives it is always the sector number.

The Sector Access Routine

Actually, now that you have seen what **SIO** requires, this subroutine (lines 9000 up) is almost self-explanatory. Once again, though, a few things need clarifying.

9230 No real reason for this, except that the resultant listing looks so much neater.

9240 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.)

9270-9320 We only allow the values we said we would. Everything else is fatal. Not fancy, but safe.

9330 A little sneaky, but we have already verified that **CMD** equals either 1 or 2, so only a legal value is possible here.

9350 The timeout value is arbitrarily large.

9360-9410 Again, we allow only legal density values. Note that 1050 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 <u>not</u> 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!

<u>Technical Sidelight</u>

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.

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 ERROR 144.

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	Total K Bytes
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
Astra
SWP
and, of course, Percom
Trak
Rana
NCT Turbo

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.

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 (heads)
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 6502 low/high order. The reason is historical: Percom uses a 680x CPU chip in their disk controller, and all 680x chips do double byte work in reverse of the 6502 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 20 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 a 4 here (five heads).

The only agreed upon values for "Density" are 0 ("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.

Reading and Writing the Config Block

As noted in Section 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's 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 auxillary 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 2.7
$304	$00	
	$06	$600, LSB first, buffer address
$306	$0F	
	$00	15, an arbitrary timeout value
$308	$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.

Our Program

The **CONFIG.BXL** program on your ToolKit disk is very long and seemingly complex. Actually, the real work is done in a couple of simple subroutines and the rest of the program is simply there to convert the raw numbers in the Config Block into readable information and/or to allow the user to easily change information in the block. Once again, then, we will resort to a description of only those parts of a program which we don't feel are self-explanatory.

1000-1200 Mostly just simple constants. Note that we will read the configuration table into the string, **Configtable$**, 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 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 <u>not</u> use the **DPEEK()** function, thanks to the fact that the double byte values are "backwards" compared to standard 6502 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 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, 1050, 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 130 tracks, 204 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 some 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.

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.)

Sequential and Other Files

Perhaps the biggest flaw in Atari DOS 2.0s (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) then 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 becomes 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).

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 some 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).

The Concept Behind PHONE.BXL, alias BlackBook

It's kind of funny that, because other DOS 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 DOS systems 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:

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 (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 64-67  Field data, as a string
```

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 BPUT here, as we shall see later.) The byte-by-byte form of an index file 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.

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.

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 **DIM**ensin 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.

15000-15280 This major routine figures out how big a file you can 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 and corresponding index. Such a lot of work for so little code! It's done with mirrors, otherwise called subroutines.

6900-7060 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 200 bytes. It does this so that it can read the sector count for this file in the directory: 200 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.

5000-5260 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).

5400-5480 **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** (**D**ata **B**ase **F**ile and inde**X**).

7500-7680 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 bytes ($FF, in fact). After performing a **NOTE** (line 7570) 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 **NOTE**d 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 out 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 all 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 modifiy 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 queue, **Pointbykey** extracts the information about sector and byte from **KEY$** and **POINT**s to the 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 **Editmenu** 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 record being inserted into all the dummy names in the index string, the 3 bytes find their way to the beginning of the string. But that is 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 now 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.

10960-11090 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 11020 and 11030). But, because we don't want to lose the **NOTE** information in that pointer, we fill it in with the standard dummy name (all $FF characters) and tack it onto the end of the index string (line 11050). We mark the record as deleted in the data file by zapping just the last character of its **PHONE$** string (11060 and 11070). 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 10240).

11300-11370 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.

9900-9940 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 keypress 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 somehow 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 "un-delete" it.

25320-25410 If the user wants to un-delete 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 something 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 **TRAP**ping at least the more 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.

2.10 MAKEAUTO

We have received many requests for this program. Its purpose is quite simple: it creates an **AUTORUN.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 **RETURN**s). 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 770 to 790) to make the job of the machine language program easier. When **AUTORUN.SYS** is loaded by DOS, your backward commands will start at location $0601, preceded by a byte containing their total length less one (line 750). 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 18: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., **POKE**s) or the device handler table (at $031A). Try it out yourself.

CHAPTER 3

BASIC XL Extended Statements

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 3.3 and 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 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 $56A,$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 **POKE**ing 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 $56A,$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.

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 -- arithmetic variable, neither a string nor an array.
Examples: TOTAL I J X0

svar -- string variable, 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 below), no parentheses may be used.

savar -- string array variable, same format, etc., as **svar** but must be a properly dimensioned array.

mvar -- matrix variable, 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 -- arithmetic expression, any valid combination of numeric values, operators, etc.
Examples: 33 7+VALUE SCORE(3*J)

* **rparm** -- receiving parameter, either an **avar** or an exclamation point followed by an **svar** or **mvar**.
Examples: TOTAL !NAMES$!VALUES()

* **cparm** -- calling parameter, either an **aexp** or an exclamation point followed by an **svar** or **mvar**.
Examples: 29*SIN(30) !TEMP$!AMAX()

slit -- string literal, a string of characters enclosed in quotation marks.
Examples: "TOTALIZE" "Test-->>"

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

* **cname** -- calling name, used to name a procedure to be CALLed, may be either an **slit** or **svar**. If an **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**).

3.3 Procedure Blocks and Related Statements

Before describing the individual statements, we present an overview of **PROCEDURE**s 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.)

```
20 Value=100
30 Min=10 : Max=90 : Gosub 100
40 Result1=Num
50 Min=10*Value : Max=90*Value : Gosub 100
60 Result2=Num
70 If Result2 > Value*Result1 Then 90
80 Print "You appear to be conservative in nature." : End
90 Print "You seem ready to take risks." : End
100 Rem THE SUBROUTINE
110 Print : Print "Please give me a number between "; Min
120 Print "   and "; 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 Result1
50 Call "Get In Range" Using 10*Temp, 90*Temp To Result2
70   If Result2 < Temp*Result1 : Type$="conservative"
80   Else : Type$="a risk taker"
90   Endif
95 Print "You seem to be "; Type$; " by nature." : End
```

[Listing continues on next page]

```
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 30, we are "Using" values of 10 and 90. But in line 100, we are "Using" the variables **Min** and **Max**. Isn't that neat? We didn't have to do the assignments to the variables before we called the subroutine: **CALL** does the work for us! It automatically moves the values (10 and 90) into the corresponding variables (**Min** and **Max**). This is called "passing parameters" to a **PROCEDURE**.

It gets better. Notice the **EXIT** statement of line 210. It specifies a value (the contents of **Temp**) which is to be placed into 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 60)? Ah, but there's line 110, with its 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 statements. 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.

3.3.1 PROCEDURE (PROC.)

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

Examples:
```
 1000 Procedure "Calculate Pay" Using Hours,Rate,!Taxtable()
  387 Procedure "Print Msg" Using !Msg$
 4040 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 the same variables.

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

```
10 Junk=20
20 CALL "Test" USING 12*17
30 Print Junk
40 End
   ...
70 PROCEDURE "Test" Using Junk
80 Print Junk+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 **P.3.**)

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 <u>is</u> 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 <u>not</u> 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 place 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 **CALL**er's parameter.

> **Technical Note:** In computer lingo, simple numeric variables are passed to a procedure block via a "call by value". Arrays and string, 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.)

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:

```
4010 CALL "Gorp" USING 33
   ...
7280 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 (0.0) 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 **CALL**ing 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 **CALL**er's variable! This can produce some bizarre results.

To demonstrate: modify line 80 of the last example program to read

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

Not surprisingly, the new **Print** in line 80 shows us that the contents of **X$** are simply "Laugh". However, look at the display resulting from line 30:

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

Do you see the problem we warned of? Changing **X$** in line 80 changed the memory at the address which **Fun$** 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 80 to read

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

Now line 30'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 20 and 90 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 **3.3.3**, 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 A$,B$ : 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 120: **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$** back to the **CALL**er, you are actually transferring the contents of **LINE$**'s variable value table entry to first **A$** and then to **B$**. But that table entry consists (among other things) of **LINE$**'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 **CALL**er'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.

3.3.2 CALL

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

```
Examples:  10 CALL "Test"
          720 CALL "Totals" USING !Values() TO Sum
          800 CALL "Get Num" TO Number
          100 CALL Proc$ USING 7,!A$ TO Result
```

The **CALL** statement has been discussed and demonstrated in both the introduction to this chapter and in the explanation of the **PROCEDURE** statement (section P.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 **CALL**ed 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 literal strings.

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

```
30 Input "Give me an index > ",Index
40 Name$=Proc$(Index;) : CALL Name$
```

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 **CALL**ing parameter (if its length is modified in the procedure block, the length change is not visible to the **CALL**er--see section P.1). 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 caution, also: **DIM**ensioned arrays need not be passed back to the **CALL**ed routine, but they must be passed in as parameters.

Secondary Considerations

The number of levels you may nest **CALL**s is limited only by the amount of **FRE**e memory left in your system which may be used by tne Run-Time Stack. Like **GOSUB**s and **WHILE**s, 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 the next chapter (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 **GOSUB**s 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 **CALL**ed procedure block, you must never attempt to **POP** the parameter variables. You can cause a system 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.

3.3.3 LOCAL

Format: LOCAL avar [,avar ...]

Examples: 730 LOCAL Temp1
1370 LOCAL Sum,N,Count,Misc

The **LOCAL** statements has been provided to allow you more flexibility in your programming. While the parameters 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 executed, 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:

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

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

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

Let's examine that program a little closer. First, line **10** is simple enough. We just assign a value to the variable and verify that it has been accepted. In line **20**, we first **GOSUB** to a routine and then again display the contents of our variable. Note that in the program's running this PRINT of **Test** is the last thing executed (other than **END**).

Line **40**, then, begins the interesting part of this program. We declare that **Test** is a **LOCAL** variables and, once again, display its value. Line **50** is a repeat of line 10 except that we assign a different value to our variable. Note that the **PRINT** verifies our change. Finally, in line **60**, we use another new statement, **EXIT**, to restore our variable to its original value, as shown by the **PRINT** in line 20.

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

Secondary Considerations

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

The fact that **LOCAL** may be used with **GOSUB**-type 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:

```
10 Input "An integer greater than 1, please >> ",N
20 Sum=0 : Gosub 50
30 Print "The sum of integers from 1 to ";N;" is ";Sum
40 End
50 Local N
60 Sum = Sum+N
70 If N=1 Then Exit
80 N=N-1 : Gosub 50
90 Exit
```

To follow what happens here, assume that we choose a value of 3 for our integer. The first time lines 50 through 70 are executed, then **Sum** will take on the value of **3** and, since **N** is not **1**, we continue on to line 80. There **N** is given a value of **2** (one less than its current value), and we again call the subroutine at line 50.

The second time through, the same things happen: **Sum** acquires a value of **5** and we do not yet do the **Exit** of line 70. In line 80, **N**'s value changes to **1** and line 50 is called once again.

This third time performing the same lines sees lines 50 and 60 performing as before, with **Sum** getting a new value of **6**. In line 70, though, since **N** now has a value of **1** we do take the **Exit**. We return to the **Gosub** of line 80, fall through to line 90, return to line 80 again, fall through to line 90 again, and (at last!) return to the original **Gosub** of line 20.

Through all of those **Exit**s, BASIC XL was keeping track of the proper value of **N** at each level, so line 30 displays accurate and sensible results for both **N** and **Sum.** Whew.

Final considerations:

Since you are still limited to 128 different variable names, in very long programs you might do well to use the same **LOCAL** variable names in all **PROCEDURE**s and subroutines. For example, you might start each such routine with a line like this:

```
3110 Local Temp1,Temp2,Temp3,Temp4
```

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

Also, since the statements built into your original BASIC XL cartridge do not understand the concept of variables being pushed onto the Run-Time stack, you must always use **Local** only at the beginning of subroutines and only in conjunction with routines ending with the **Exit** keyword. In particular, never try to **POP** a variable which has been made **Local.**

3.3.4 EXIT

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

Examples:
```
390 EXIT 10*Maxvalue
799 EXIT Flag,!Names$
24990 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 3.3, 3.3.2, and 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).

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 looks for 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 the 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 **Call**ers 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 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 many **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 4 for discussions of the example programs provided on your ToolKit disk.

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 3.4.1 and 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 **SORTDOWN** 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 show 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 before "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 ; 3,5
40 For I=1 To 5 : Print Array$(I;) : Next I
50 Run
```

Once again, enter some strings in response to **INPUT**'s prompt. This time, though, pay special attention to the third through fifth characters of each string. Notice anything funny about the sorted order? That's right, it is based solely on the characters in those positions. If you have worked with BASIC 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 **Array$** in that same line shows, the semi-colon is always necessary when referring to an element of a string array.)

Now, since the **SORTUP** of line 30 refers to the entire 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 30 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 want 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 30. 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 will 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 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.

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.)

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 10 ; 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.)

CHAPTER 4

Example BASIC XL Programs with Extended Statements

This chapter gives examples of programs written using the extended statements presented in Chapter 3. Three of the programs here (those in Sections 4.1, 4.2, and 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 Chapter 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 Chapter 2 for details on other parts of these programs.

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 3.3, you probably aren't too surprised to find the names 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 280 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 290. 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 280. 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 280, **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 290, when **Number** had a value of 2. Okay, so we return back to that same line 290, and **Result** gets a value of **1**. Then we continue on to line 300, where we **Exit** with what?

Well, we just said **Result** is **1**, and since **Number** had a value of **2** when line 290 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 **Call**s on that last **Exit**, so this time we end up back at line 290 from the time we **Call**ed with **Number** equal to **3**, and **Result** gets a value of **2**. By the same logic, we continue to line 300 and **Exit** with **3*2**.

This time, though, we have dispensed with all the **Call**s except the original one, in line 190, 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 succesively smaller values until the value reaches **1**. Then start **Exit**ing 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 100). Then try finding out what **100!** 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 **5000!** 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.)

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 string array can represent.

100-240 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 **FILE$()** 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 **DIM**ensions.

260, 680 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 goo programming exercise for you to us.

270-340 This is an easy way to read in the directory. The **LINE$** 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 semicolon. 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?

580-640 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.

4.3 SORTNUM.BXE

In the presentation of the sort statements in section 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 100 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 fortituitous 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 **SORTUP** 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 $00? Well, it is now time to examine the program listing to see how we turned adversity to advantage.

150-160 The only reason for the **DUMMY$** string is to provide an address for that single element numeric array. Recall that in BASIC XL (and Atari BASIC), string and array variables **always** use memory in the order they are **DIM**ensioned. Thus the address of **VALUE** has to be one greater than the address of **DUMMY$**.

180 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 20 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.

280, 360 We're going to generate, manipulate, and display 20 random numbers.

290 This is just to give each element of the array a **LEN**gth of six. Otherwise, the sort process won't know how many bytes in each array element need sorting.

300 We generate random numbers in an arbitrary range, but one which is easy to view.

310-320 See how we move the six bytes of the floating point 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 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 20 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.

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.

4.5 DISKIO.BXE

Another fairly simple conversion from the original standard BASIC XL program. This time, though, there is a 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 **LEN**gth (like **BGET**, direct sector access doesn't change the length of a string). Similarly, we put a **RETURN** character into the string (line 50) 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$(11,10)

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 10 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*(10+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.)

4.6 PHONE.BXE

This last program "conversion" is our "Little Black Book" program from Section 2.9. It was a monster as a standard program. It remains a monster using extended statements. But, perhaps, it is a more managable 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 5000 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 5000 to 5260. 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.